8-2012

AIR: Adaptive Dynamic Precision Iterative Refinement

Jun Kyu Lee
jlee57@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Jun Kyu Lee entitled "AIR: Adaptive Dynamic Precision Iterative Refinement." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Itamar Arel, Robert J. Hinde, Robert J. Harrison

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by JunKyu Lee entitled "AIR: Adaptive Dynamic Precision Iterative Refinement." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Dr. Gregory D. Peterson
Major Professor

We have read this dissertation
and recommend its acceptance:

Dr. Itamar Arel

Dr. Robert J. Hinde

Dr. Robert J. Harrison

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

AIR: Adaptive Dynamic Precision Iterative Refinement

A Dissertation Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

JunKyu Lee
August 2012

Dedication

This dissertation is dedicated to my parents, GilSub Lee and MiHee Kim.

Acknowledgements

First of all, I would like to thank my major advisor Dr. Gregory D. Peterson for his constant support and his greatly insightful advice for the overall of this work and especially for FPGA computing and data-driven architecture in this work. I would like to thank Dr. Robert J. Harrison for giving me valuable suggestions for linear algebra and power aware computing in this work. I would like to thank Dr. Robert J. Hinde for giving me interest of computational science by his amazing teaching for the computational science class. I would like to thank Dr. Itamar Arel for serving on my committee. Also, I would like to thank Dr. Don Bouldin for providing me one of the best classes to learn FPGA design and Xing Fu for helping me to measure the power consumption on the FPGAs. I would like to thank my previous and current research colleagues, Dr. Akila Gothandaraman, Dr. Junquing Sun, Dr. Depeng Yang, Dr. Saumil Merchant, Getao Liang, David Jenkins, Rick Weber, Yu Du, Gwanghee Son, Susan Gao, Rui Ma, Kiran Kasichayanula and Yeting Feng for their help and encouragement. I would like to thank National Science Foundation to partially support this work. This work was partially supported by the National Science Foundation grant CHE-0625598. Finally, I would like to thank my family, GilSub Lee, MiHee Kim, and SooYoung Lee for supporting me mentally to keep on doing this research. Most of all, I would like to thank God for everything.

.

Abstract

In high performance computing, applications often require very accurate solutions while minimizing runtimes and power consumption. Improving the ratio of the number of logic gates implementing floating point arithmetic operations to the total number of logic gates enables greater efficiency, potentially with higher performance and lower power consumption. Software executing on the fixed hardware in Von-Neuman architectures faces limitations on improving this ratio, since processors require extensive supporting logic to fetch and decode instructions while employing arithmetic units with statically defined precision. This dissertation explores novel approaches to improve computing architectures for linear system applications not only by designing application-specific hardware but also by optimizing precision by applying adaptive dynamic precision iterative refinement (AIR). This dissertation shows that AIR is numerically stable and well behaved. Theoretically, AIR can produce up to 3 times speedup over mixed precision iterative refinement on FPGAs. Implementing an AIR prototype for the refinement procedure on a Xilinx XC6VSX475T FPGA results in an estimated around 0.5, 8, and 2 times improvement for the time-, clock-, and energy-based performance per iteration compared to mixed precision iterative refinement on the Nvidia Tesla C2075 GPU, when a user requires a prescribed accuracy between single and double precision. AIR using FPGAs can produce beyond double precision accuracy effectively, while CPUs or GPUs need software help causing substantial overhead.

Table of Contents

List of Tables

List of Figures

List of Notations

A	original matrix
q	relative forward error in the solution in direct method
ϵ_i	applied precision for step i
ϵ_A	original precision used to store the elements of matrix A
ϵ_L	lower precision than $\boldsymbol{\epsilon_A}$
ϵ_H	higher precision than $\boldsymbol{\epsilon_A}$
ϵ_{AR}	arbitrary precision
$\epsilon_j^{(i)}$	applied precision for step j at i^{th} iteration for AIR
n	matrix size
β	increasing ratio for the run time according to mantissa bit width increase
P	growing factor from LU factorization
x^*	exact solution
x_c, $x^{(i)}$	computed solution, computed solution at i^{th} iteration
$\delta x^{(i)}$	absolute error in the computed solution at i^{th} iteration
$\delta w^{(i)}$	accumulated round-off error in steps 2 and 4 at i^{th} iteration
c_i	constant depending on matrix size
m	required number of iteration
C	effective capacitance
Ω	convergence rate
$\Omega_f^{(i)}$	convergence rate in forward error at i^{th} iteration
$\Omega_f^{(i)}$	convergence rate in backward error at i^{th} iteration
$\Omega_{QR/LU}$	convergence rate for QR and LU in mixed precision iterative refinement
$\|\cdot\|_\infty$	Infinity norm
$\|\cdot\|_2$	2 norm
$\kappa(A)_\infty$	infinity norm condition number
$\kappa(A)_2$	2 norm condition number
$\kappa(A)_{SK}$	Skeel's condition number
$\#_{COMP}$	number of transistors participating in floating point arithmetic operations
$\#_{SUP}$	number of transistors participating in supporting floating point operations
$\tilde{\alpha}$	total number of transistors participating in floating point operations, in other words, $\#_{COMP} + \#_{SUP}$
Λ	design effectivenss, in other words $\#_{COMP}/(\#_{COMP} + \#_{SUP})$
t	mantissa bit width
$t^{(i)}$	required mantissa bit width in steps 2 and 4 at i^{th} iteration in AIR
t_f	required mantissa bit width in steps 2 and 4 at final iteration in AIR
t_L	required mantissa bit width for lower precision
t_{L_LU}	required mantissa bit width for lower precision for LU
t_{L_QR}	required mantissa bit width for lower precision for QR
t_A	mantissa bit width for original precision
t_H	mantissa bit width higher than the original precision

List of Notations (Continued)

T	run-time
T(t)	run-time depending on a mantissa bit width
T(ϵ)	run-time depending on a precision
T_{LU}	run-time in mixed precision iterative refinement employing LU
T_{QR}	run-time in mixed precision iterative refinement employing QR
γ	ratio of mantissa bit width between the initial precision and refinement precision, t_L/t_f
AT1	accelerator type 1 employing statically defined precision ALUs: multi-cores and GPUs.
AT2	accelerator type 2 employing dynamically defined arbitrary precision ALUs: FPGAs
D	dynamic power consumption
S	static power consumption
V	operational voltage for the computing cores
U	total power consumption. In other words, D +S
Flops	number of floating point operations
P	time-based performance: Flops/second
F_{CLK}	clock-based performance: Flops/clock-cycle
F_{JLE}	energy-based performance: Flops/joule considering total power consumption. In other words, P/U
F_{JLE_D}	energy-based performance: Flops/joule considering dynamic power consumption. In other words, P/D

Chapter 1

Introduction

"Everything should be made as simple as possible, but not simpler."

Even though we do not know who made this quote, people guess that this great quote was coined by Albert Einstein [1]. Years later, a composer mentioned this quote could be applied to music [2]. Here we note that this quote can be applied to High Performance Computing (HPC) too and this dissertation applies the quote to HPC, in particular to linear algebra applications. HPC emphasizes high accuracy, extreme performance, and lower power consumption (energy efficiency). Of these, the most important consideration by far is the solution accuracy: if a solution is not accurate enough, the performed computation is useless. Hence, if the computation successfully produces a prescribed solution accuracy, the next consideration may be either higher performance or energy efficiency. With HPC, maximizing performance is the typical focus. If one cares more about the costs of running the HPC system, energy efficiency may begin to rival high performance in importance. The question is how we can produce a prescribed accuracy with higher performance and lower power consumption. This research seeks to find an answer to this question with an approach in the spirit of the quote above.

First, to produce a prescribed accuracy, we need to choose an algorithm to implement the assigned application on a computing platform. The achievable accuracy from an algorithm depends on the system condition. Hence, we need to choose *the simplest algorithm* in terms of the runtime with respect to the system condition so the prescribed accuracy is obtained. Second, to produce higher performance and lower power

consumption or better energy efficiency, we need to choose an appropriate computing platform and effectively implement the algorithm on this computing platform. We expect the best computing platform will implement the necessary operations and precision, but with a minimal amount of overhead that reduces performance and wastes energy. Moreover, the metrics on which we will focus primary attention will include runtime and energy efficiency, subject to constraints on the required resulting numeric accuracy and on power. The runtime will be expressed using wall-clock time in seconds, and the energy efficiency will be expressed using floating point operations per joule.

The computational precision impacts the resulting accuracy, performance, and energy efficiency. Figure 1 describes the impact of precision. Since a lower precision Arithmetic Logic Unit (ALU) is smaller than a higher precision ALU, relatively larger numbers of lower precision ALUs can be employed within the same area as for a few higher precision ALUs. Based on Figure 1, if we employ a lower precision for the computation, we can utilize smaller ALUs so that the parallelism for the computation can be increased with the additional ALUs that can be fit in an area. Also, lower precision ALUs can be implemented using shorter pipelines and wires so that the applied clock rate can be improved. Hence, employing lower precision ALUs can achieve higher performance. The number of transistors in a lower precision ALU is less than for a higher precision ALU. Hence, employing a lower precision ALU for the computation consumes less power than employing a higher precision ALU. Therefore, employing the *least sufficient precision* that produces the prescribed solution accuracy can result in higher performance with lower power consumption.

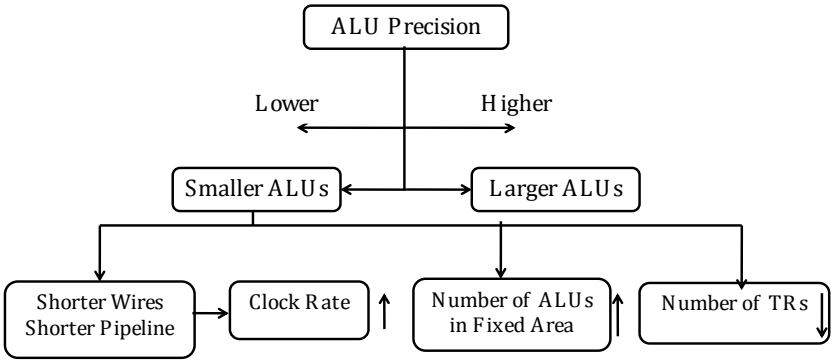

Figure 1. Impact of precison

VLSI technology has developed rapidly, and the computational bit-width has been increasing along with the demand for higher accuracy [3]. Parallel computational platforms such as multi-core processors or General Purpose Graphical Processing Units (GPUs) are now becoming prevalent, but these platforms utilize statically defined floating point ALUs, often with support for only single and double precision [4, 5]. Another approach is to employ Field Programmable Gate Arrays (FPGAs) for parallel computation [6]. FPGAs are a data-driven architecture supporting bit-level programming for computation unlike Von-Neumann architectures. Therefore, we can obtain effectively higher performance and lower power consumption on the FPGAs not only because we can optimize the FPGA circuit by employing application-specific approaches with the *least sufficient logic gates* to perform necessary floating point operations but also because FPGAs can employ an arbitrary precision for the floating point operations so that we can fully exploit this flexible precision for higher performance and lower power consumption

based on Figure 1 [7-9]. However, the choice of exploitable precision has limits in most computing platforms such as microprocessors and GPUs because they employ either single or double precision in hardware.

Linear system solvers are popular applications in scientific computation [10] and appear in many applications such as signal processing, electromagnetic simulation, quantum scattering, spectrophotometry and the least-squares problem [11-14]. Linear systems can be solved either by the direct or iterative method [15, 16]. The direct method (DIR) requires a matrix decomposition such as GEPP (Gaussian Elimination with Partial Pivoting), QR (Orthogonal and Upper triangular matrix decomposition), or the Cholesky method, each having $O(n^3)$ complexity, where *n* is the matrix size. Iterative methods such as GMRES (Generalized Minimal Residual method) or CG (Conjugate Gradient method) do not require a matrix decomposition and have $O(n^2)$ complexity. DIR is usually used for dense matrix applications and the iterative method is used for sparse matrix applications. Large dense linear systems become important [14] and iterative refinement [17] can be useful for such large dense linear system applications to produce an accurate solution. However, iterative refinement has limitations in terms of achievable accuracy and performance, if ALUs with statically defined precision are employed.

At this point, we may consider arbitrary precision ALUs. Many computer scientists or engineers wonder if arbitrary precision computation is practical [18, 19] and previous research investigated computer architectures supporting arbitrary precision computation (i.e. dataflow architecture [20, 21] or reconfigurable computing [22]). Arbitrary precision computations have been investigated since some applications require

highly accurate numeric solutions [23, 24]. These high precision arithmetic operations were typically supported by software routines, but now FPGAs can support arbitrary precision ALU circuits in hardware.

Dynamically defined arbitrary precision computation is revitalized in this dissertation with the case study of iterative refinement on FPGAs. This dissertation proposes the Adaptive Dynamic Precision Iterative Refinement (AIR) algorithm, which employs the *least sufficient precisions* for iterative refinement on FPGAs. Hence, this dissertation discusses the AIR algorithm employing the *least sufficient precisions* and the implementation design for AIR employing the associated *least sufficient logic gates* on FPGAs in order to produce a prescribed accuracy in the solution for linear systems, resulting in effectively higher performance and lower power consumption.

Chapter 2 discusses related work for iterative refinement, categorizes the iterative refinements according to precision usage, and suggests employing a higher precision for the iterative refinement triangular system solver step when triangular matrices possess large condition numbers. Chapter 3 discusses a sufficient condition in which GEPP is a superior choice to QR for iterative refinement in terms of runtime performance given a prescribed accuracy. Chapter 4 performs an average case study for iterative refinement convergence with respect to system conditions. Chapter 5 introduces the AIR algorithm, its correctness, and estimated runtime, and presents numeric experiments for the correctness of AIR. Chapter 6 describes the AIR refinement procedure implementation on the XUPV5-LX110T FPGA development board [25], performance evaluation of AIR in terms of time-, clock-, and energy-based performance, and introduces the design

effectiveness which can explain energy efficiency for computation. Chapter 7 finally concludes the dissertation and discusses future work.

The main contribution in this dissertation is to introduce a practically applicable adaptive dynamic precision algorithm AIR on a real computing platform and examine its performance and energy efficiency for the algorithm. As for additional contributions:

1. This dissertation provides a good tutorial for iterative refinement by categorizing possible iterative refinement approaches according to choices of precision.

2. This dissertation provides a practically sufficient condition for mixed precision iterative refinement in which LU is a superior choice to QR in terms of runtime given an accuracy, matrix, and computing platform.

3. Theoretical convergence rates for iterative refinements are often far from actual convergence rates for real applications. Therefore, this dissertation explores empirical convergence rates for mixed precision iterative refinements for average cases.

4. This dissertation introduces design effectiveness, which can explain energy efficiency for FPGA computing.

Chapter 2

Iterative refinement

Heretofore, the previous literature on iterative refinement made it difficult for non-experts in numeric analysis to understand the detailed behavior of iterative refinement such as the impact of precision on iterative refinement for achievable accuracy and convergence rate [7-9, 26-36]. The field also lacks a clear taxonomy to categorize iterative refinement approaches effectively.

This chapter provides a tutorial to understand iterative refinement. In section 2-1, we compare iterative refinement to a water purifier to help readers understand the behavior of iterative refinement. We categorize possible iterative refinement approaches according to various ways to apply precision. According to these categories, we discuss their convergence rates, achievable accuracy and runtime by adapting the numeric analyses of Jankowski and Wozniakowski [31].

2-1. Overview of iterative refinement

This section helps reader to understand the idea of iterative refinement before we get into performing quantitative analysis. Algorithm I describes iterative refinement. We will often use the terminology of steps 1, 2, 3, 4, and 5 to specify the steps for iterative refinement in this chapter. We compare the Algorithm I to a water purifier since we believe that a water purifier is a good example to describe iterative refinement mechanism clearly.

Algorithm I. Iterative refinement :

Given: A : a square matrix (n × n)
i = 0;

Step 1:	GEPP LU w/partial pivoting	$O(n^3)$, precision ϵ_1
Step 2:	$LU \times x^{(1)} = P \times b$	$O(n^2)$, precision ϵ_2
Step 3:	$r^{(i)} = A \times x^{(i)} - b$	$O(n^2)$, precision ϵ_3
Termination:	if $\|\|r^{(i)}\|\|$ is small enough: backward error	
Step 4:	$LU \times z^{(i)} = P \times r^{(i)}$	$O(n^2)$, precision ϵ_4
Step 5:	$x^{(i+1)} = x^{(i)} - z^{(i)}$	$O(n)$, precision ϵ_5
Termination:	if $\|\|x^{(i+1)}\|\|$ is accurate enough: forward error	

Go back to Step3 until $\|\|r^{(i)}\|\|$ or $\|\|x^{(i+1)}\|\|$ is accurate enough

Note.

GEPP: Gaussian Elimination with Partial Pivoting
P: Permutation matrix from GEPP
ϵ_i: Precision applied at step i,
r: Residual vector, x: Solution,
b: Right side vector in system equation (Ax = b)

Figure 2 describes the water purifier depicting the iterative refinement mechanism. In essence, one can picture a process for filtering impurities out of water as a sequence of filtration steps. In the first step, a coarse-grained filter is applied, resulting in larger impurities being removed from the water. Next, additional filtration steps can be repeatedly performed using finer-grained filters to progressively remove more of the impurities until sufficiently pure water is obtained (the size or magnitude of the impurities is below some threshold).

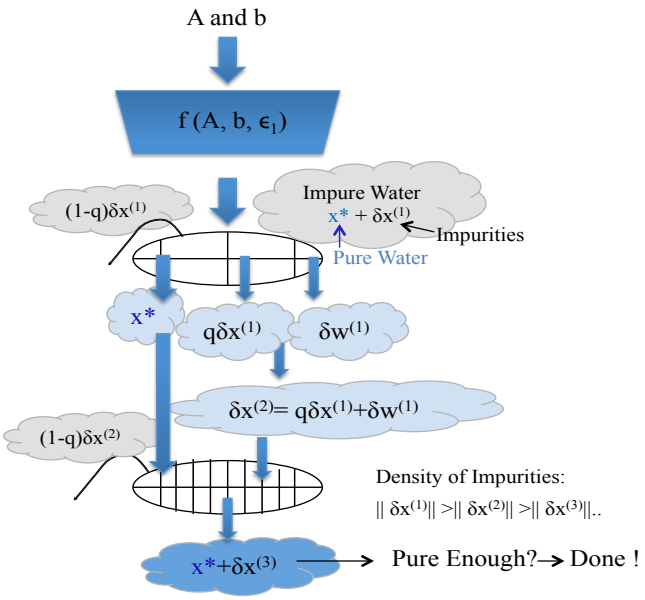

Figure 2. Mixed precision iterative refinement mechanism

A similar approach is used with iterative refinement, with an approximate solution first computed that has some error. A sequence of refinement steps can then be applied that progressively reduce the magnitude of the error until the solution is sufficiently accurate.

Steps 1 and 2, the approximate solution calculation in Algorithm I, corresponds to the blue square box in Figure 2. At the end of step 2, the system produces an approximate solution with an error $\delta x^{(1)}$. In step 3, a residual is sought that contains some round-off error; therefore, the residual is not entirely correct. Some percentage of $\delta x^{(1)}$ (i.e., $(1 - q)\ \delta x^{(1)}$) is eliminated in the first iteration, but the remainder of $\delta x^{(1)}$ (i.e., $q\delta x^{(1)}$) and the round-off errors in steps 3 and 5 (i.e., $\delta w^{(1)}$) are passed through the first filtration mesh (i.e., the first iteration) and transferred to the second filtration mesh (i.e., the second

iteration). However, the round-off errors in steps 3 and 5 could be negligible in practice, if iterative refinement employs sufficiently high precision in steps 3 and 5 (e.g., doubled precision in steps 3 and 5 compared to the precision in steps 1, 2, and 4, as discussed later in chapter 4). The implementation of steps 1, 2, and 4 in Algorithm I determines the value of q. If either higher precision is employed in steps 1, 2, and 4 or a numerically stabler algorithm (i.e., Householder QR) is employed in steps 1, 2, and 4, we can generally obtain a smaller value for q, but it may cause performance degradation due to the cost of higher precision arithmetic for the computationally intensive work $O(n^3)$ of matrix decomposition or for a more stable algorithm. Hence, $\delta x^{(2)}$ represents the quantity of error after the first iteration; therefore it contains some fraction of $\delta x^{(1)}$ (i.e., $q\ \delta x^{(1)}$) along with round-off errors from steps 3 and 5 (i.e., $\delta w^{(1)}$) in Figure 2. The size of $\delta x^{(2)}$ is smaller than $\delta x^{(1)}$ when q is smaller than unity, which is the success condition of iterative refinement. The applied precision in steps 3 and 5 generates noise, $\|\delta w^{(j)}\|$ affecting the convergence rates (i.e., if "infinite precision" is applied in steps 3 and 5, the noise disappears). This iterative process continues until the solution is accurate enough.

We discuss iterative refinement by comparing it to a water purifier to understand the iterative refinement mechanism more clearly. Impure water in Figure 2 (i.e., computed solution) becomes purer (i.e., becomes closer to the exact solution) as it experiences more stages (i.e., more iterations). The required number of stages (i.e., required number of iterations) depends on the filtration mesh size at each stage (i.e., convergence rate) and the desired purity of water (i.e., prescribed accuracy for the solution). If the filtration mesh size is smaller (i.e., good convergence rate), it is possible

to obtain pure water quickly. The filtration mesh size is related with the processing speed for iterative refinement; the convergence rate depends on the condition number of the matrix and the precision applied in steps 2 and 4. The desired water purity depends on the precisions in steps 3 and 5 in Algorithm I. In terms of implementation for iterative refinement, the filtration mesh size and desired water purity (i.e., arithmetic precision and prescribed accuracy) are not controllable in most computing platforms (e.g. microprocessors, GPUs, or Cell processors), since most computing platforms employ statically defined precisions when implementing their arithmetic units. In FPGAs, both the filtration mesh size (i.e. arithmetic precision) and the desired water purity (i.e., achievable accuracy) are controllable, since FPGAs can support arbitrary precision circuits. Having presented a high level overview of iterative refinement and the importance of precision, the next section addresses previous work related to the research in this dissertation.

2-2. Related work

Wilkinson noticed the direct method is not always capable of producing a reliable solution for systems of equations and devised the iterative refinement approach in 1948 to ameliorate this issue [17]. From this point forward in this dissertation, we refer to this iterative refinement approach as Wilkinson's Iterative Refinement (WIR). WIR employs a higher precision than the original precision in step 3 and the original precision in steps 1, 2, 4, and 5. The main idea of WIR is to remove the effects of degrading accuracy due to the condition number by employing a higher precision in step 3. Wilkinson pointed out

that only the residual calculation is sensitive to numeric error, so it is sufficient to provide a higher precision for step 3 only. Wilkinson presented his correctness proof using block floating arithmetic for WIR in 1963 [36]. In block floating point arithmetic, a block of data shares an exponent for arithmetic operations so that they can perform fixed-point arithmetic in the block. Therefore, the computed data during the computation do not have to rearrange the exponent and fraction parts in the block in block floating-point arithmetic [22, 36].

Moler investigated WIR using floating point arithmetic in 1967 [37] and pointed out that the main difference for numeric analyses for iterative refinement between fixed point and floating point comes from the round off errors in seeking a residual vector when subtracting two nearly equal floating point numbers. In fixed-point arithmetic, the residual calculation error of higher precision is not considered since fixed-point arithmetic just addresses the truncation error when the computed value of the residual is truncated to lower precision for step 4. The analyses in [36, 37] show that WIR is able to produce the original precision accuracy in forward error if the system is not too ill conditioned.

Since 1948, various iterative refinement approaches have been derived based on WIR employing various precisions according to the computing steps. In 1977, Jankowski and Wozniakowski proposed that any method to solve linear systems can be numerically stable and well behaved when using iterative refinement even if the method employs the original precision in step 3 as long as the relative error in steps 2 and 4 is less than unity and the system is not ill-conditioned [31]. The special skill in the numeric analysis

performed in [31] considers the relative error in solution in steps 2 and 4 (i.e. q in [31]) and applies it to the numeric analysis. Therefore, the analysis can be applied to iterative refinement employing any method for steps 2 and 4. For practical purposes, the error bounds depending on matrix sizes are represented using some constant rather than a specific bound in [31]. The numeric analysis in [31] is a useful tool to understand the practically useful properties for iterative refinement since q relates to the properties of the direct method and errors from matrix decomposition often do not have special meanings.

Motivated by the work of [31], Skeel explored the numeric behavior for iterative refinement in 1980 if GEPP is used in steps 2 and 4 (c.f., any method is used in steps 2 and 4 in [31]) and the original precision is employed in step 3 [34]. In his numeric analysis, he separately put $O(\epsilon^2)$ so that the analysis can be more simpler and practically useful. He concludes that iterative refinement can remove the effects of improper scaling for Gaussian elimination. Skeel concluded that iterative refinement along with a numerically stable algorithm enhances the stability in a strong sense [34]. In 1991, Higham investigated the same case as with [34], but QR was employed in steps 2 and 4 instead of GEPP [28]. Higham also showed that iterative refinement employing QR is able to eliminate the effects of improper scaling. Since iterative refinement in [28, 31, 34] employs the original precision in step 3, we denote this iterative refinement method as the Original Precision Iterative Refinement (OPIR). In particular, the iterative refinement in [28, 34] utilizes only one precision (i.e., the original precision) in the entire iteration process so we denote this iterative refinement method as Fixed Precision Iterative Refinement (FPIR) (e.g., same terminology used in [30]).

In 1973, Stewart suggested that utilizing a higher precision for step 5 can improve the quality of the solution even more than the iterative refinement proposed by Wilkinson [35, 38]. In his analysis, Stewart uses the relative errors of the residual vectors rather than the applied precisions in order to obtain a qualitative understanding of the iterative refinement. He introduced two terms for successful convergence. One term is related to a condition number and initial precision, already proposed by Moler, while the second term is related to the relative errors for the residual vectors. Based on his analysis, the relative errors for the residual vectors should be less than the reciprocal of 2 times the condition number for successful convergence.

In 1981, Kielbasinski pointed out that employing arbitrary precision for refinement can result in the desired accuracy with $O(n^2)$ computation complexity if the applications require extremely high precision accuracy [18]. He called his algorithm Binary Cascade Iterative Refinement (BCIR). The BCIR algorithm has important differences from Algorithm I. BCIR refines both solution and correction vectors to achieve the desired numeric accuracy. However, BCIR is not a practical algorithm for real applications since it requires the condition number before computations [39]. Because of these difficulties, another paper [19] proposes using BCIR while employing doubled-mantissa arithmetic. This approach increases the precision per iteration with doubled precision compared to the previous precision. The authors claim that the algorithm is more practical than the original BCIR since they can describe the accuracy and the potential performance in terms of a given initial precision. However, BCIR in [19] is also not a practical algorithm, since it requires knowing the required number of

iterations before computation and in practice the required number of iterations cannot be predicted before computation.

In 2005, Demmel et al proposed Extra-Precise Iterative Refinement (EPIR) which employs doubled precision in step 5 so that iterative refinement can produce a more accurate solution [27, 40]. In [27], EPIR behaviors are categorized into three regions known as strong convergence, weak convergence, and no convergence according to the condition numbers. The authors in [27] mentioned that the iterative refinement is the same as the Newton method and employing doubled precision in step 5 has impact on accuracy when the system is ill scaled.

In 2006, Lanjou et al proposed iterative refinement employing single precision (i.e., lower precision) in steps 1, 2, and 4 and double precision (i.e., the original precision) in steps 3 and 5 [32]. We denote this iterative refinement method as Single and Double Precision Iterative Refinement (SDIR) from this point forward. SDIR is able to obtain single precision performance without losing double precision accuracy. The idea of SDIR is that as long as the relative error in steps 2 and 4 is less than 1 and the system is not ill conditioned, SDIR is able to produce the original precision accuracy in backward error. In SDIR, reducing the precision to single precision for the matrix decomposition does not affect the solution accuracy if the matrix is not too ill conditioned and the original precision (i.e., double precision) accuracy can be obtained back by a refinement procedure using the original precision. For example, in the case of a linear system solver consisting of double precision matrix and vector elements, the authors

suggested performing matrix factorization using single precision and applying double precision in steps 3 and 5. SDIR is discussed further in [32, 41, 42].

In 2008, Sun et al suggested employing an arbitrary lower precision for LU decomposition on FPGAs and applying double precision in steps 3 and 5 to obtain a higher performance than SDIR without losing double precision accuracy [9]. We denote this iterative refinement method as Arbitrary Initial Precision Mixed Precision Iterative Refinement (AMIR) from this point forward in this dissertation. In 2011, Lee and Peterson recommended employing arbitrary precisions on FPGAs for all the iterative refinement steps so that a user can achieve an arbitrary precision accuracy with effectively high performance [33]. We call this iterative refinement method eXtended Mixed Precision Iterative Refinement (XMIR) hereafter.

Now, we organize the various iterative refinement methods with respect to the utilization of precision. To organize the iterative refinement methods according to the utilization of precision, we first define two types of iterative refinement methods in terms of achievable accuracy, known as Higher Precision Iterative Refinement (HPIR) and Original Precision Iterative Refinement (OPIR). HPIR employs a higher precision than the original precision in any step and OPIR employs the original precision in steps 3 and 5. Hence, WIR [17, 43] and BCIR [18, 19] are categorized as members of the HPIR class and FDIR [28, 34], SDIR [32], and AMIR [9] are categorized as members of the OPIR class. XMIR [33] is categorized in both OPIR and HPIR classes according to the implementation choices since XMIR employs arbitrary precisions in steps 3 and 5 that could fit either HPIR or OPIR. Next, we also clarify iterative refinement in terms of the

number of applied precisions. We refer an iterative refinement as Mixed Precision Iterative Refinement (MPIR) if the iterative refinement employs multiple statically defined choices of precision. WIR, SDIR, AMIR, EPIR, and XMIR are categorized in the MPIR class. We refer to an iterative refinement method as belonging to the Arbitrary Precision Iterative Refinement (APIR) class if the iterative refinement method employs arbitrary numbers of precisions for iterative refinement. BCIR is categorized in the APIR class. Note that APIR could be well suitable on FPGAs, since FPGAs can support arbitrary precision circuits in hardware. A new APIR algorithm is proposed in chapter 5, which is Adaptive Dynamic Precision Iterative Refinement (AIR). AIR can be categorized to HPIR, OPIR, and APIR. Figure 3 represents the categorization of iterative refinement mehtods from previous work according to the utilization of different precisions. In order to see the details of mathematical proofs for correctness for iterative refinements, please refer to [30, 31, 34-38].

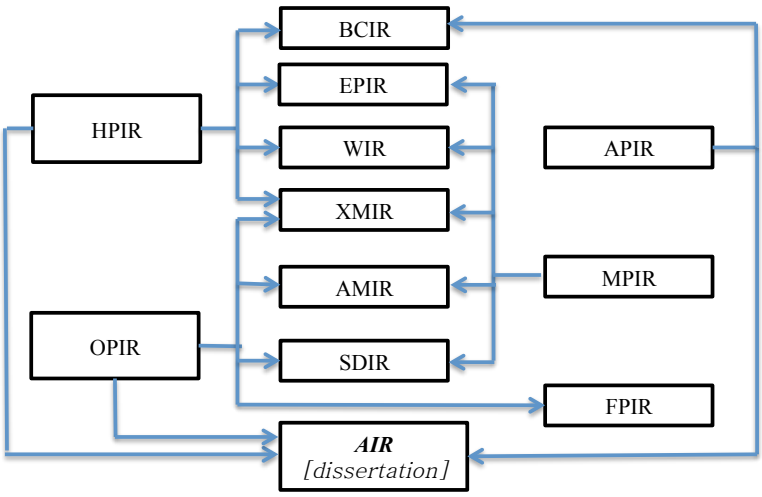

Figure 3. Iterative refinement methods categorized by precisions

In terms of implementation efforts, Wilkinson implemented WIR on the Automatic Computing Engine [17]. The algorithm proposed in [32] was implemented using accelerators such as multi-core processors, GPUs, CELL processors, and FPGAs [9, 32, 41, 42, 44-47]. The MPIR produced approximately ten times better performance than double precision FPIR in the Cell processors without losing double precision accuracy [41, 42, 47]. The MPIR was implemented in MAGMA v0.2 (Matrix Algebra on GPUs and Multicore Architectures v0.2) on multi-core processors and GPUs [44]. Additional research [45, 46] discussed implementation of iterative refinement using GPUs and multi-cores. In [45], matrix decomposition is performed on the GPU and the refinement procedure is performed on the host. MPIR employing Conjugate Gradient (CG) was implemented in FPGAs for model predictive control applications and the authors report that 26X speedup can be achievable compared to a 3GHz CPU [48]. In [46], the possible hardware combinations of modern computing platforms to implement MPIR were discussed. In [9], AMIR was implemented on the Cray XD-1. The AMIR employed arbitrary precision for step 1 on FPGAs and double precision for the steps 3 and 5 on the host. The AMIR on the Xilinx Virtex-4 FPGA can achieve approximately 40 GFlops when the matrix is not ill conditioned. The applicable initial precision for AMIR was further studied according to various condition numbers in [8, 49]. Variable precision arithmetic on FPGAs is discussed in [50-52]. Based on the ability to utilize precision on a computing platform, applicable computing platforms for different types of iterative refinement methods can be described as Table I.

TABLE I. APPLICABLE COMPUTING PLATFORMS

⊙: VERY PRACTICAL, O: PRACTICAL, Δ: NOT PRACTICAL, ×: NOT RECOMMENDED

		Multicore(1)	(1)+GPUs	(1)+FPGAs
DIR		O	O	O
WIR [36], [37]		Δ	Δ	Δ
SIR [34], [31]		O	O	O
MPIR	SDIR [32], [44]	O	⊙	Δ
	AMIR[9, 46]	×	×	⊙
	XMIR [33]	×	×	⊙
BCIR [18, 19, 53]		×	×	Δ
AIR [dissertation]		×	×	⊙

2-3. Impact of precision on iterative refinement

It is hard to see the impact of choice of precision on convergence rates and accuracy for iterative refinement methods from the previous literature, since the previous numeric analyses do not distinguish the precision in each step. Hence, we performed numeric analysis to see the impact of precision in each step on the iterative refinement convergence rate and accuracy. We explain the impact in Figure 2. Our numeric analysis adapts the analysis performed in [31].

Since iterative refinement focuses on the quantities of errors, the norm wise interpretation is appropriate. The norm could be considered as the size of a vector [16] and we employ $\|\cdot\|$ notation for infinity norm from this point forward. Hence, $\delta x^{(2)}$ in Figure 2 can be represented as $\|\delta x^{(2)}\| \leq q \|\delta x^{(1)}\| + \|\delta w^{(1)}\|$.

Letting $\Omega_{Mf} = \max_j \{ \Omega_f{}^{(j)} : \Omega_f{}^{(j)} \geq q + \|\delta w^{(j)}\|/\|\delta x^{(j)}\| \}$, (2.1)

the norm wise error bound at k^{th} iteration is represented as follows:

$|| \delta x^{(k+1)} || \leq \Omega_f{}^{(k)} || \delta x^{(k)} || \leq \Omega_{Mf} || \delta x^{(k)} ||$ where $\delta x^{(k+1)}$ is the error of computed solution and $\Omega_f{}^{(k)}$ is a convergence rate at the k^{th} iteration.

Using recursive relation,

$||\delta x^{(k+1)}|| \leq \Omega_{Mf}{}^{k} ||\delta x^{(1)}|| \leq q\, \Omega_{Mf}{}^{k} ||x^*|| \leq \Omega_{Mf}{}^{k+1} ||x^*||$. Finally, $||\delta x^{(k+1)}||/||x^*|| \leq \Omega_{Mf}{}^{k+1}$.

Based on (2.1), the convergence rate Ω_{Mf} consists of the relative error in the solution q from steps 2 and 4 and the round off errors in steps 3 and 5 of the current iteration. The relative error in solution q in steps 2 and 4 is inherently determined from triangular system solvers and does not vary significantly. However, when the iteration proceeds, the second term $||\delta w^{(j)}||/||\delta x^{(j)}||$ in Ω_{Mf} becomes larger since the computed errors become smaller since $||\delta x^{(j)}||$ becomes smaller as the iteration proceeds. The size of $||\delta w^{(j)}||$ generally stays unless the iterative refinement employs dynamically changing precisions. We explore what constituents are in $||\delta w^{(j)}||$ using the numeric analysis performed in [31].

In step 2, $r^{(i)} = (I + \delta I_4{}^{(i)}) (b - Ax^{(i)} + \delta y^{(i)}) = (I + \delta I_4{}^{(i)}) (-A\delta x^{(i)} + \delta y^{(i)})$,

$|| r^{(i)} || \leq (1 + \epsilon_4) (||A|| ||\delta x^{(i)}|| + || \delta y^{(i)}||)$,

where $||\delta y^{(i)}|| \leq c_1 ||A|| ||x^{(i)}|| \epsilon_3 + O(\epsilon^2)$, $c_1 = c_2 + ||\delta x^{(i)}||/||x^{(i)}||$, c_2 is a constant depending on matrix size, and $\delta I_4{}^{(i)}$ is a diagonal matrix whose infinity norm bound is $||\delta I_4{}^{(i)}|| \leq \epsilon_4$.

In step 3, $(A + \Delta A) (z^{*(i)} + \delta z^{(i)}) = r^{(i)}$, where

$z^{(i)*} = A^{-1} r^{(i)}$

$= (-\delta x^{(i)} + A^{-1} \delta y^{(i)}) + A^{-1}\, \delta I_4{}^{(i)} (-A\delta x^{(i)} + \delta y^{(i)})$, $||\delta z^{(i)}|| \leq q\, ||z^{*(i)}||$, and $q < 1$. (2.2)

In step 4,

$x^{(i+1)} = x^{(i)} + z^{(i)} + \delta d^{(i)} = x^* + \delta x^{(i)} + z^{(i)*} + \delta z^{(i)} + \delta d^{(i)}$

$= x^* + A^{-1}\delta y^{(i)} + A^{-1}\,\delta I_4^{(i)}\,(-A\delta x^{(i)} + \delta y^{(i)}) + \delta z^{(i)} + \delta d^{(i)} = x^* + \delta x^{(i+1)}$

where $\delta x^{(i+1)} = A^{-1}\delta y^{(i)} + A^{-1}\,\delta I_4^{(i)}(-A\delta x^{(i)} + \delta y^{(i)}) + \delta z^{(i)} + \delta d^{(i)}$ (2.3)

and $\|\delta d^{(i)}\| \le \| x^* + \delta x^{(i)} + z^{(i)*} + \delta z^{(i)} \|\, \epsilon_5$. Therefore,

$$\| \delta x^{(i+1)} \| \le \|A^{-1}\|\, \|\delta y^{(i)}\| + \epsilon_4 \|A^{-1}\|\, \| -A\delta x^{(i)} + \delta y^{(i)} \| + q\, \|z^{*(i)}\| + \|\delta d^{(i)}\|$$

$\le \epsilon_2\, k(A)\, \|\delta x^{(i)}\| + (1+\epsilon_4)\, \|A^{-1}\|\, \|\delta y^{(i)}\| + (q(1+\epsilon_5)+\epsilon_5)\, \|z^{*(i)}\| + \epsilon_5 \| x^* + \delta x^{(i)} \|$

$\le ((q(1+\epsilon_5)+\epsilon_5)\,(1+\epsilon_4 k(A)) + \epsilon_4\, k(A))\, \|\delta x^{(i)}\| +$

$[(1+\epsilon_4)(\,1+(q(1+\epsilon_5)+\epsilon_5)\,)\, c_1\, k(A)\epsilon_3 + \epsilon_5\,]\, \| x^* + \delta x^{(i)}\|$

$= (\,q + (1+q)(\epsilon_5 + k(A)\epsilon_4)\,)\, \|\delta x^{(i)}\| + (\, c_1\, k(A)(1+q)\,\epsilon_3 + \epsilon_5)\, \| x^* + \delta x^{(i)}\| + O(\epsilon^2),$

$= (q + k(A)\,(1+q)\,(\epsilon_4 + c_1\epsilon_3) + (2+q)\,\epsilon_5)\, \|\delta x^{(i)}\| + (\, c_1\, k(A)(1+q)\,\epsilon_3 + \epsilon_5)\, \| x^*\| + O(\epsilon^2).$

$= q\, \|\delta x^{(i)}\| + \|\delta w^{(i)}\|.$ (2.4)

Hence,

$\|\delta w^{(i)}\| = (k(A)\,(1+q)\,(\epsilon_4 + c_1\epsilon_3) + (2+q)\,\epsilon_5)\|\delta x^{(i)}\| + (c_1 k(A)(1+q)\epsilon_3 + \epsilon_5)\, \| x^*\| + O(\epsilon^2).$ (2.5)

Theorem 2.1 (Th 2.1): If the system is not too ill conditioned,

Achievable accuracy in forward error : $AC_f \le \Omega_{f2}/(1-\Omega_{f1})$, (Th 2.1.1)

Achievable accuracy in backward error : $AC_b \le \Omega_{b2}/(1-\Omega_{b1})$, (Th 2.1.2)

Convergence rate at i^{th} iteration for forward error : $\Omega_{f1} + \Omega_{f2}\|x^*\|/\|\delta x^{(i)}\|$, (Th 2.1.3)

Convergence rate at i^{th} iteration for backward error :

$$\Omega_{b1}^{(i-1)} + \Omega_{b2}^{(i-1)}\, \|A\|\, \|x^{(i-1)}\|/\|b - Ax^{(i-1)}\|$$ (Th 2.1.4)

where $\Omega_{f1} = q + k(A)\,(1+q)\,(\epsilon_4 + c_1\epsilon_3) + (2+q)\,\epsilon_5$, $\Omega_{f2} = c_1\, k(A)(1+q)\,\epsilon_3 + \epsilon_5 + O(\epsilon^2)$,

$\Omega_{b1}^{(i)} = \{(1+qk(A))\epsilon_4 + q\, \|A\|\, \|\delta x^{(i)}\|/\|b - A\, x^{(i)}\|\}\, \zeta^{(i)}$, $\Omega_{b2}^{(i)} = (\epsilon_5 + c_1^{(i)}\, \epsilon_3(1+q\, k(A)))\, \zeta^{(i)}$

$+ O(\epsilon^2)$, $\Omega_{b1} = \mathrm{Max}_i(\Omega_{b1}^{(i)})$, $\Omega_{b2} = \mathrm{Max}_i\,(\Omega_{b2}^{(i)})$, and $\zeta^{(i)} = \|x^{(i)}\| / \|x^{(i+1)}\|$.

Proof: From (2.4) and (2.5), $||\delta x^{(i+1)}||/|| x^*|| \le (q + k(A)\ (1 + q)\ (\epsilon_4 + c_1\epsilon_3) + (2+q)\ \epsilon_5)\ ||\delta x^{(i)}||/|| x^*|| + (c_1\, k(A)(1 + q)\ \epsilon_3 + \epsilon_5) + O(\epsilon^2)$.

Letting $\Omega_{f1} = q+k(A)\ (1 + q)\ (\epsilon_4 + c_1\epsilon_3) + (2+q)\ \epsilon_5$ and $\Omega_{f2} = c_1\, k(A)(1 + q)\ \epsilon_3 + \epsilon_5 + O(\epsilon^2)$,

$$||\delta x^{(i+1)}||/|| x^*|| \le \Omega_{f1}||\delta x^{(i)}||/|| x^*|| + \Omega_{f2} \le \Omega_{f1}\ (\Omega_1||\delta x^{(i-1)}||/|| x^*|| + \Omega_{f2}) + \Omega_{f2}$$

$$\le \Omega_{f1}{}^{i}\, ||\delta x^{(1)}||/||x^*|| + \Omega_{f2}/(1-\Omega_{f1}). \qquad (2.6)$$

Hence, ϵ_4 is a sensitive factor for the convergence rate and is recommended as a higher precision than the reciprocal of the condition number. The precision ϵ_4 in Ω_{f1} is caused by the truncation from steps 3 to step 4. Letting $\Omega_f^{(j)}$ as $\Omega_f^{(j)} = \Omega_{f1} + \Omega_{f2}||x^*||/||\delta x^{(j)}||$, (2.7)

Ω_{f1} reduces the relative errors and $\Omega_f^{(j)}$ reduces the absolute errors in computed solution. Hence, the proof for (Th 2.1.3) is complete.

From (2.6), $\lim_{i\to\infty} ||\delta x^{(i+1)}||/||x^*|| = \Omega_{f2}/(1-\Omega_{f1})$. Hence, the proof of (Th 2.1.1) is complete. As another approach, when iterative refinement does not converge any longer (i.e., $\Omega_f^{(j)} \approx 1$), the final accuracy is as follows from (2.7) : $||\delta x^{(j)}|| /||x^*|| \approx \Omega_{f2} / (1 - \Omega_{f1})$.

We explore backward error analyses for iterative refinement.

$|| b - Ax^{(i+1)}|| \le ||A\delta x^{(i+1)}||$.

From (2.3),

$|| b - Ax^{(i+1)} || \le || \delta y^{(i)} + \delta I_4^{(i)}(-A\delta x^{(i)} + \delta y^{(i)}) + A\delta z^{(i)} + A\delta d^{(i)} ||$,

From (2.2),

$\le c_1^{(i)}\, ||A||\, ||x^{(i)}||\, \epsilon_3 + \delta I_4^{(i)}(-A\delta x^{(i)}) + q\, ||A||\, || A^{-1}r^{(i)} || + A\delta d^{(i)} + O(\epsilon^2)$

$\le c_1^{(i)}\, ||A||\, ||x^{(i)}||\, \epsilon_3 + \epsilon_4|| A\ \delta x^{(i)}|| + q\, ||A||\, ||-\delta x^{(i)} + A^{-1}\delta y^{(i)} + A^{-1}\ \delta I_4^{(i)}\ (-A\delta x^{(i)} + \delta y^{(i)})|| + ||$

$A\delta d^{(i)} || + O(\epsilon^2)$

$\leq (\epsilon_5 + c_1^{(i)} \epsilon_3)\ ||A||\ ||x^{(i)}|| + \epsilon_4 ||\ A\delta x^{(i)}|| + q\ ||A||\ ||\delta x^{(i)}|| + qk(A)||\delta y^{(i)}|| + q\ k(A)\ \epsilon_4\ ||A\delta x^{(i)}|| +$

$(1+q)\ \epsilon_5\ ||A||\ ||\ z^{(i)*}|| + O(\epsilon^2)$

$\leq\ (\epsilon_5 + c_1^{(i)}\ \epsilon_3(1+ q\ k(A)))||A||\ ||x^{(i)}|| + (1+ q\ k(A))\ \epsilon_4\ ||\ A\delta x^{(i)}||\ + q\ ||A||\ ||\delta x^{(i)}|| + O(\epsilon^2),$

Hence,

$||b - Ax^{(i+1)}||/\ ||A||\ ||x^{(i+1)}|| \leq (\epsilon_5 + c_1^{(i)}\ \epsilon_3(1+ q\ k(A)))\ ||x^{(i)}||/||x^{(i+1)}|| + (1+qk(A))\epsilon_4\ ||b - A\ x^{(i)}\ ||\ /||A||\ ||x^{(i+1)}||\ + q\ ||\delta x^{(i)}||\ /\ ||x^{(i+1)}|| + O(\epsilon^2),$

$\leq (1 + qk(A))\epsilon_4\ ||b - A\ x^{(i)}\ ||\ /||A||||x^{(i)}||\ (||x^{(i)}||/||x^{(i+1)}||) + q\ ||\delta x^{(i)}||\ /\ ||x^{(i+1)}|| + (\epsilon_5 + c_1^{(i)}\ \epsilon_3(1+ q\ k(A)))\ ||x^{(i)}||/||x^{(i+1)}|| + O(\epsilon^2),$

$\leq\ ||b\ -\ A\ x^{(i)}\ ||\ /||A||\ ||x^{(i)}||\ \{(1+qk(A))\epsilon_4||x^{(i)}||/||x^{(i+1)}||\ +\ q\ ||A||\ ||\delta x^{(i)}||/||b\ -\ A\ x^{(i)}\ ||\ (||x^{(i)}||/||x^{(i+1)}||)\ \}\ +\ (\epsilon_5 + c_1^{(i)}\ \epsilon_3(1+ q\ k(A)))\ ||x^{(i)}||/||x^{(i+1)}||\ +\ O(\epsilon^2),$

Letting $\Omega_{b1}^{(i)} = \{(1+qk(A))\epsilon_4 + q\ ||A||\ ||\delta x^{(i)}||/||b - A\ x^{(i)}\ ||\}\ \zeta^{(i)}$, $\Omega_{b2}^{(i)} = (\epsilon_5 + c_1^{(i)}\ \epsilon_3(1+ q\ k(A)))\ \zeta^{(i)} + O(\epsilon^2)$, and $\zeta^{(i)} = ||x^{(i)}||\ /||x^{(i+1)}||$,

$$||b - Ax^{(i+1)}||/(||A||\ ||x^{(i+1)}||) \leq \Omega_{b1}^{(i)}\ ||b - Ax^{(i)}||/||A||\ ||x^{(i)}|| + \Omega_{b2}^{(i)}, \tag{2.8}$$

$$||b - Ax^{(i+1)}||/(||A||\ ||x^{(i+1)}||) \leq (\Omega_b^{(i)} + \Omega_{b2}^{(i)}\ ||A||\ ||x^{(i)}||/||b - Ax^{(i)}||)\ ||b - Ax^{(i)}||/||A||\ ||x^{(i)}||.$$

Hence, the proof for (Th 2.1.4) is complete.

Setting $||A||\ ||\delta x^{(i)}|| \leq c_b^{(i)}\ ||b\ -\ A\ x^{(i)}\ ||$ where $c_b \geq 1$, $\Omega_{b1}^{(i)} = \{(1+qk(A))\epsilon_4 + c_b^{(i)}q\}\ \zeta^{(i)}$.

Letting $\Omega_{b1} = \max_i \Omega_{b1}^{(i)}$ and $\Omega_{b2} = \max_i \Omega_{b2}^{(i)}$,

$$||b - Ax^{(i+1)}||/(||A||\ ||x^{(i+1)}||) \leq\ \Omega_{b1}^{\ i}\ ||b - Ax^{(1)}||/||A||\ ||x^{(1)}|| + \Omega_{b2}/(1 - \Omega_{b1}). \tag{2.9}$$

$\lim_{i\to\infty} ||b\ -\ Ax^{(i+1)}||/(||A||\ \ ||x^{(i+1)}||) = \Omega_{b2}/(1\ -\ \Omega_{b1})$. Hence the proof for (Th 2.1.3) is complete. Q. E. D.

Property 2.1 (P 2.1): The iterative refinement converges if the system is not too ill-conditioned and $||\delta x^{(j)}||/|| x^*|| > (c_1\, k(A)(1 + q)\, \epsilon_3 + \epsilon_5)/(1 - q)$

Proof: From (2.4) and (2.5), it should be satisfied that $q + ||\delta y^{(j)}||/||\delta x^{(j)}|| < 1$ to make convergence successful. If the system is not too ill conditioned, the term $k(A)\,(1 + q)\,(\epsilon_4 + c_1\epsilon_3) + (2 + q)\,\epsilon_5$ is considerably smaller than unity. Hence,

$(c_1\, k(A)(1 + q)\, \epsilon_3 + \epsilon_5)\, || x^*||/||\delta x^{(j)}|| < (1 - q).$ Q. E. D.

Remark: Based on (P 2.1), if the size of $||\delta x_j||$ is less than $(c_1\, k(A)(1 + q)\, \epsilon_3 + \epsilon_5)\, ||x^*||/(1 - q)$, the convergence may not occur any more. If an iterative refinement changes the precisions for steps 3 and 5 dynamically, it can obtain an arbitrary accuracy (discussed in chapter 5). In practice, q is relatively smaller than unity (e.g., at least one decimal digit convergence). Hence, the precisions in steps 3 and 5 can control accuracy since they are related to the number of iterations rather than convergence rate. The precision for step 3 is recommended to use a higher precision than step 5 since the condition number is related with the precision for step 3.

Property 2.2 (P 2.2): The convergence becomes worse when iteration proceeds.

Proof: $\Omega^{(j)}$ becomes larger since $|| x^*||/||\delta x^{(j)}||$ becomes larger as long as $\Omega^{(j)} < 1$. Q. E. D.

One may wonder about the case in which the error $||\delta x^{(j)}||$ is zero so that $\Omega_f^{(j)}$ is theoretically infinity. The following property has the answer.

Property 2.3 (P 2.3): If the iterative refinement produces an exact solution, the next iteration produce a computed solution having the error bound as follows: $||\delta x^{(i+1)}||/||x^*|| \leq \Omega_{f2} + O(\epsilon^2)$.

Proof: By letting $\delta x^{(j)} = 0$ from (2.2), we can obtain the accuracy as follow :

$$|| \delta x^{(i+1)} || \leq ||A^{-1}||\, ||\delta y^{(i)}|| + \epsilon_4 ||A^{-1}||\, || \delta y^{(i)} || + q\, ||z^{*(i)}|| + ||\delta d^{(i)}||$$

$\leq (1+\epsilon_4)\, ||A^{-1}||\, ||\delta y^{(i)}|| + (q(1+\epsilon_5)+\epsilon_5)\, ||z^{*(i)}|| + \epsilon_5 || x^* ||$

$\leq (c_1\epsilon_3\, k(A)\, (1+\epsilon_4) + \epsilon_5)\, ||x^*|| + c_1\epsilon_3\, k(A)\, (q(1+\epsilon_5)+\epsilon_5)\, ||x^*|| + O(\epsilon^2)$

$= (c_1\epsilon_3\, k(A)\, (1+q)) + \epsilon_5)\, ||x^*|| + O(\epsilon^2) = \Omega_2\, ||x^*|| + O(\epsilon^2)$. Q. E. D.

Remark: Even though the convergence rate $\Omega_f^{(j)}$ is infinity, the relative error in the solution does not diverge based on (2.1), since the previous error is zero.

Property 2.4 (P 2.4): If the precision ϵ_3 in step 3 employs approximately $\epsilon_5/k(A)$ (i.e., if the precision ϵ_5 in step 5 employs approximately $k(A)\, \epsilon_3$), the accuracy of the precision of step 5 can be achieved.

Remark: The property of (P 2.4) may explain why EPIR is able to converge more than WIR in some ill-conditioned systems (refer to Figure 13 in [40]). Demmel et al compared the accuracies of WIR to IR employing the doubled precision of the original precision in step 5 in ill-conditioned matrices in Figure 13 in [40]. EPIR is able to converge a few iterations faster than WIR so that the accuracy becomes improved. The convergence property is represented in the property of (P 2.1). If in either a well conditioned system or a too ill conditioned system, $c_1\, k(A)(1 + q)\, \epsilon_3$ in (P 2.1) is relatively smaller or larger

than ϵ_5, then EPIR is not much more useful than WIR. In some conditioned systems, c_1 k(A)(1 + q) ϵ_3 in (P 2.1) could be comparably equal to ϵ_5, so employing higher precision in step 5 has some benefit.

Table II shows the comparison of the methods in terms of accuracy and theoretical runtime. We discuss the theoretical runtime in (A 3.2) in chapter 3. In the usual cases, we ignore the runtime from step 3 to 5. In unusual cases, the runtime from 3 to 5 is dominant (i.e., a prescribed accuracy is extremely high compared to the applied lower precision). The m represents the required number of iterations.

TABLE II. COMPARISON FOR ITERATIVE REFINEMENTS

F: Forwared error, B: Backward error

		Precisions (Algorithm I)			Run time		Accuracy
		$\epsilon_{1,2,4}$	ϵ_3	ϵ_5	Usual	Unusual	
DIR		ϵ_A	None	None	$n^3T(\epsilon_A)$	$n^3T(\epsilon_A)$	$\kappa(A)\epsilon_A$ (F)
FPIR		ϵ_A	ϵ_A	ϵ_A	$n^3T(\epsilon_A)$	$n^3T(\epsilon_A)$	$\kappa(A)\epsilon_A$ (F) ϵ_A (B)
MPIR	WIR	ϵ_A	ϵ_H	ϵ_A	$n^3T(\epsilon_A)$	$2n^2mT(\epsilon_H)$	ϵ_A (F)
	EPIR	ϵ_A	ϵ_H	ϵ_A or ϵ_A^2	$n^3T(\epsilon_A)$	$2n^2mT(\epsilon_H)$	ϵ_A (F)
	SDIR	ϵ_S	ϵ_D	ϵ_D	$n^3T(\epsilon_L)$	$4n^2mT(\epsilon_A)$	ϵ_A (B)
	AMIR	ϵ_L	ϵ_D	ϵ_D	$n^3T(\epsilon_L)$	$4n^2mT(\epsilon_A)$	ϵ_A (B)
	XMIR	ϵ_L	ϵ_H or ϵ_A	ϵ_H or ϵ_A	$n^3T(\epsilon_L)$	$4n^2mT(\epsilon_{A/H})$	ϵ_A (F/B)
APIR	BCIR	ϵ_L	$\epsilon_{AR}^{(i)}$	$\epsilon_{AR}^{(i)}$	$n^3T(\epsilon_L)$	$4n^2T(\epsilon_H)$	ϵ_A (F)
	AIR	ϵ_L	$\epsilon_{AR}^{(i)}$	$\epsilon_{AR}^{(i)}$	$n^3T(\epsilon_L)$	$4n^2T(\epsilon_H)$	ϵ_A (F/B)

2-4. Convergence rates

The basic idea of iterative refinement is to exploit lower precision computation for the computationally intensive work $O(n^3)$ in step 1, since the original precision accuracy can be regained by employing a higher precision accuracy with comparably cheaper $O(n^2)$ computational work. As a consequence, one of the main concerns with iterative refinement is successful convergence. As long as there exists convergence property in iterative refinement, the convergence rate is not that important in practice. For example, if we apply a higher precision for matrix decomposition, the convergence rate will be improved, but there will be performance loss due to the $O(n^3)$ for matrix decomposition with the higher precision. Convergence rates are also related to which method we applied for matrix decomposition since the backward error in solving linear systems using DIR depends on the matrix decomposition algorithm. Both DIR and an iterative method can be used for steps 2 and 4, as long as the relative error in the solution q in steps 2 and 4 is relatively smaller than unity [31]. Iterative refinement with DIR requires a matrix factorization (i.e., step 1), but iterative refinement with an iterative method does not perform step 1. We consider iterative refinement with DIR in this dissertation.

DIR is originally designed to solve dense linear systems using matrix decomposition methods such as LU, QR, and the Cholesky method. DIR utilizes only one precision to solve linear systems. The Cholesky method is used only when the system matrix is hermitian (e.g., symmetric). Both LU and QR can be used to solve regular dense matrices. LU is more widely used than QR in step 1 since LU requires less computational

work than QR [5-8]. The number of floating point operations are $n^3/3$ for Cholesky matrix decomposition, $2n^3/3$ for LU, and $4n^3/3$ for QR. To solve triangular system equations, n^2 operations are required for Cholesky, $2n^2$ for LU, and $3n^2$ for QR. Hence, the runtime for iterative refinement with the direct method is mainly determined by matrix factorization.

Based on the analysis [36], we are able to see the convergence rate in terms of backward error in steps 2 and 4 and the condition number of the matrix. Wilkinson assumes that the rounding errors come only from matrix decomposition. We employ the same assumption. Considering LU decomposition for matrix factorization for step 1, the rounding errors from the matrix decomposition in step 1 can be represented as E in (2.10). $L \times U = A + E$ (2.10), where L is the lower triangular matrix, U is the upper triangular matrix, A is a matrix before the matrix decomposition, and E is the backward error which makes (2.10) hold. The norm of E is generally getting smaller by increasing the precision for LU decomposition. The lower and upper triangular matrices obtained from step 1 are used for steps 2 and 4. After the matrix decomposition, steps 3 and 5 can be described as (2.11) and (2.12). Finally, we could obtain forward error analysis as in (2.11) and backward error analysis as in (2.12).

$$r^{(i)} = b - Ax^{(i)} \tag{2.11}$$

$$x^{(i+1)} = x^{(i)} + (L \times U)^{-1} r^{(i)} \tag{2.12}$$

$$x^{(i+1)} - x = x^{(i)} - x + (L \times U)^{-1} A(x - x^{(i)})$$

$$= [I - (L \times U)^{-1} A](x^{(i)} - x) = [I - (L \times U)^{-1} A]^i (x^{(1)} - x) \tag{2.13}$$

$$r^{(i+1)} = A(x^{(i+1)} - x) = A[I - (L \times U)^{-1} A]^i (x^{(1)} - x)$$

Based on (2.13), "$[I - (L\times U)^{-1}A]^i$" approaches the zero matrix if $L\times U$ is an excellent preconditioner which means $(L\times U)^{-1}A$ is very near to the identity matrix. The convergence rates can be described as $[I - (L\times U)^{-1}A]$ (i.e., q in Figure 2 since it does not consider round-off errors in steps 3 and 5). In other words, the convergence rates depend on how close $(L\times U)^{-1}A$ could be to the identity matrix. That is also why the condition number of the matrix and the backward error are important for successful convergence. The proof for successful convergence in terms of condition numbers and backward errors is described as follows:

$$L\times U = A + E = A(I + A^{-1}E)$$

Letting $F = I - (LU)^{-1}A = I - (A+E)^{-1}A = I - (A(I + A^{-1}E))^{-1}A = I - (I+A^{-1}E)^{-1}$

$= (I+A^{-1}E)\,(I+A^{-1}E)^{-1} - (I+A^{-1}E)^{-1} = (I+A^{-1}E - I)\,(I+A^{-1}E)^{-1} = A^{-1}E\,(I+A^{-1}E)^{-1}$, (2.14)

the norm of matrix *F* should be less than 1 for the success condition [4]. Therefore,

$$||F|| = ||\, A^{-1}E\,(I+A^{-1}E)^{-1}|| \le ||\, A^{-1}E\,||\;||(I+A^{-1}E)^{-1}|| < 1$$

where $||\cdot||$ is a matrix sub-ordinate infinity norm [6]. Based on (2.14), if the backward error is smaller, the convergence can be improved. If $(I + A^{-1}E)$ is a non singular matrix, $(I + A^{-1}E)$ can be described as follows:

$$(I + A^{-1}E)^{-1} = (I + A^{-1}E - A^{-1}E)\,(I + A^{-1}E)^{-1}$$

$= (I + A^{-1}E)\,(I + A^{-1}E)^{-1} - A^{-1}E\,(I + A^{-1}E)^{-1} = I - A^{-1}E\,(I + A^{-1}E)^{-1}$. Hence,

$$||(I + A^{-1}E)^{-1}|| \le ||I|| + ||A^{-1}E||\;||(I + A^{-1}E)^{-1}||,$$

$$(1 - ||A^{-1}E||)\;||(I + A^{-1}E)^{-1}|| \le ||I|| = 1.$$

If $||A^{-1}E|| < 1$, $||(I + A^{-1}E)^{-1}|| \le (1 - ||A^{-1}E||)^{-1}$. Therefore,

$$||F|| = ||\, A^{-1}E\,(I+A^{-1}E)^{-1}|| \le ||\, A^{-1}E\,||\;||(I+A^{-1}E)^{-1}||$$

$$\leq \| A^{-1}E \| / (1 - \|A^{-1}E\|) \leq (\|A^{-1}\| \|E\|) / (1 - \|A^{-1}\| \|E\|) < 1. \quad (2.15)$$

Consequently, $(\|A^{-1}\| \|E\|) < ½$. The success condition in terms of condition number and backward error is: $(\|E\|/\|A\|) (\|A\| \|A^{-1}\|) < ½$. Therefore, the matrix condition number should be less than one half of the reciprocal of the relative backward error for success for any kind of iterative refinement, which is a sufficient condition for success for iterative refinement if there exists no rounding errors in steps 3 and 5. Even though the condition is violated, the iterative refinement may converge. The relative back error $\|E\|/\|A\|$ mainly depends on the precision we applied for matrix decomposition and the matrix decomposition algorithm.

If we apply a higher precision for matrix decomposition then the relative backward error generally becomes smaller. If the relative backward error is large, the allowable condition number for the mixed precision solver should be small. The relative error not only depends on the precision level but also which algorithm we apply such as LU, Cholesky, or QR. The stabilities for LU, Cholesky, or QR are discussed in [4-6, 8, 29]. Relative backward errors may vary depending on some other factors such as growth factors (the ratio between largest elements of the decomposed matrix and original matrix), so some researchers have sought to characterize the relative backward errors using statistical approaches [8, 23]. Iterative refinement works well in the case L×U is an excellent preconditioner (Generally, it is!), as long as LU decomposition employs a pivoting strategy (e.g., partial pivoting).

LU performed by GEPP is commonly used for iterative refinement. Good scaling is important for GEPP since it can change the decision for pivot selection in partial

pivoting [54]. Otherwise, the scaling does not influence accuracy even though it can reduce the condition number [54]. The impact of partial pivoting on accuracy is very well explained geometrically in [54-56]. In [54-56], the authors mention that partial pivoting helps to make the upper triangular matrix pivot oriented. They explain the stability of GEPP using a standard coordinate. They view Gaussian Elimination (GE) as parallelizing each row vector in a hyper-plane with standard coordinates. For example, a 2 x 2 matrix consists of two row vectors. Along with a right side vector b, the matrix forms two lines in the x-y standard coordinates. In GE, the second line is parallelized with the x axis, since the x component of the upper triangular matrix is zero. Hence, by partial pivoting the first line is more x-axis oriented than the second line within a column in the matrix. In other words, it is highly probable to make the two lines near orthogonal with respect to each other, so that small perturbation does not affect the intercept significantly compared with non-pivoting. Since partial pivoting searches for a pivot within a column, it is not guaranteed the pivot is the largest element in its row. Complete pivoting can provide a more pivot oriented upper triangular matrix than partial pivoting in a geometrical sense. Hence, finding a solution using the upper triangular matrix obtained from GE employing any pivoting strategy is more stable than without a pivoting strategy. Transforming the system of equations ($Ax = b$) to an upper triangular system ($Ux = b'$, where $b'=L^{-1}b$) does not cause significant rounding off errors compared to solving the triangular systems. Hence using a higher precision for solving triangular systems might be useful if the matrix is not well properly scaled or GE does not employ any pivoting strategy (i.e., condition numbers of triangular matrices are high). We introduce another MPIR method

using a mixed precision direct method in the next section in order to handle extremely ill scaled systems possessing large condition numbers for triangular matrices. Lower triangular matrices from GEPP are diagonally dominant (i.e., the diagonal element is larger or equal than the other elements in each row). Hence, the triangular system solver from lower triangular matrices does not produce significant round-off errors. If we do not employ a partial pivoting strategy, the element sizes in both L and U matrices are arbitrary. As a consequence, the arbitrary distribution of elements on triangular matrices can cause instability in the solution. This geometrical interpretation in [54-56] can be extended to QR decomposition. Since the Q matrix (especially from Householder) is close to orthogonal among the row vectors, seeking an intermediate vector (i.e., $b'=Q^T b$) using the Q matrix is stable. The round-off errors to solve a linear system using QR mainly come from seeking a final solution using the upper triangular matrix. If the upper triangular matrix is near orthogonal among its row vectors, the solution is reliable. Due to round off errors, Householder QR and Gram-Schmidt do not produce exact orthogonal matrices. Householder QR forms an orthogonal matrix having a better orthogonality than Gram-Schmidt since Householder QR employs orthogonal triangularization and Gram-Schmidt employs triangular orthogonalization [16]. In Householder QR, suppose Householder QR produces an exact orthogonal matrix. Then, solving triangular systems using Householder QR is unconditionally stable [57, 58] since the 2 norm condition number of upper triangular system is the same as the original matrix condition number as follows : $|| A || = ||QR|| = || R ||$, since Q just rotates the vector Rx, where x is an arbitrary vector. Hence, $k(A) = ||A|| \, ||A^{-1}|| = ||QR|| \, ||R^{-1}Q^{-1}|| = ||R|| \, ||R^{-1}||$. (2.16)

2-5. Higher precision in solving triangular systems

GEPP sometimes produces large growth factor because GEPP searches a column to find the largest pivot. In such cases, the condition numbers of triangular systems can be extremely large because the systems are far from pivot-oriented systems [54, 56, 59] and iterative refinement employing GEPP in step 1 may not produce successful convergence. Employing GE with rook pivoting [59] or complete pivoting in step 1 may handle such cases, but iterative refinement with GEPP can also handle them effectively by employing a higher precision in steps 2 and 4. We name this iterative refinement Higher Precision in Triangular System Solvers with MPIR (HTIR). HTIR employs original or higher precision in solving triangular systems. Hence, the precision to solve triangular systems in steps 2 and 4 is higher precision than the precision applied for matrix factorization.

We verify this iterative refinement method with the boundary value example shown in [60] by setting n = 91, k = 1, L = 60, and C = 6. We test 100 cases by changing input vector b. We compare HTIR with SDIR. HTIR employs double precision in solving triangular system, while SDIR employs single precision. Hence, the required runtime for HTIR is comparable to SDIR since HTIR employs higher precision than SDIR for computationally inexpensive work $O(n^2)$. Based on MATLAB condition estimation, the system has $k(A)_2 \approx 133$, $k(L)_2 \approx 3.7 \times 10^9$, growth factor $\approx 1.4 \times 10^{26}$, and $k(U)_2 \approx$ inf. In this case, HTIR shows 100% successful convergence and SDIR shows 8% successful convergence.

2-6. Summary of chapter 2

In this chapter, we first describe the iterative refinement mechanism qualitatively by comparing it with a water purifier. Iterative refinement reduces errors in each step if the relative error in the solution in steps 2 and 4 is less than unity but it produces some round off errors from steps 3 and 5. Hence, employing higher precision in comparably cheap computational work (e.g., steps 2, 3, 4, and 5) is a promising idea for iterative refinement.

WIR is the first proposed iterative refinement method in order to obtain a better accuracy in forward error. Based on WIR, EPIR was proposed and it showed some good influence to accuracy in forward error compared to WIR. To our knowledge, BCIR was the first effort to obtain higher performance by employing lower precision for computationally expensive work in iterative refinement as long as the relative error in the solution in steps 2 and 4 is less than unity. With a similar idea, various iterative refinement methods such as SDIR, AMIR, and XMIR were proposed.

Since refinement procedure consists of three steps (i.e., steps 3, 4 and 5) and can employ three types of precisions (e.g., ϵ_L, ϵ_H, and ϵ_A) each step, 27 types of iterative refinements can be theoretically possible. However, we will discuss the practically useful combinations of iterative refinement out of the 27 types. We remove bad combinations for practical reasons. First of all, decomposing a matrix with higher precision is not a good idea, since the error from matrix decomposition is relatively small (refer to SWOP stages in [56]) and it requires most computationally work in iterative refinement. Second, employing lower precisions in steps 3 and 5 does not make sense. In such cases, the

round off errors from steps 3 and 5 are significant so that iterative refinement does not converge to the solution. Third, if the original precision is employed for matrix decomposition, employing lower precision for triangular system solver is not a good choice, since it causes degrading convergence rate. Finally, employing a higher precision in step 5 compared to the precision in step 3 is not a good choice, since the precision in step 3 is much more sensitive to the condition number for accuracy. Hence, the possible combinations of iterative refinements are reduced to 14. Figure 4 shows the possible combinations. Based on Figure 4, each iterative refinement method has its own characteristic.

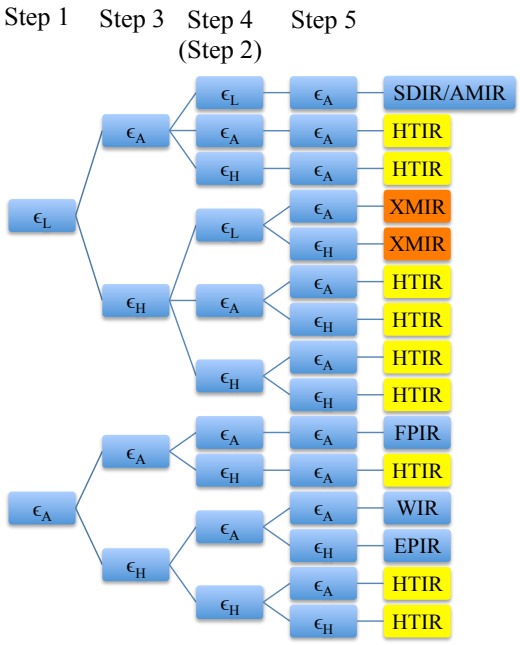

Figure 4. Iterative refinements with precisions

For example, SDIR is a good method if the condition number of matrix does not exceed the reciprocal of single precision, AMIR is a good method if the system is well conditioned, WIR is a good method if a user wants an original accuracy in backward error and the condition number of the system does not exceed the reciprocal of the original precision, and FPIR is a good method if the system is well conditioned and a numerically stable algorithm is employed in steps 2 and 4. In this chapter, we explored the remaining area for iterative refinement methods employing statically pre-defined precisions, which has not been explored in previous work, which is the main contribution of this chapter.

Chapter 3

Mixed precision iterative refinement with LU and QR

From this point forward, please refer to Algorithm II for iterative refinement algorithm so that we can discuss iterative refinement with fewer steps (i.e., 4 steps), since steps 2 and 4 are identical process in Algorithm I. GEPP (i.e. LU with partial pivoting) is widely used in MPIR, but large growth factors from LU may make the convergence fail. QR is numerically more stable than LU but requires more computational work. The best choice between LU and QR to solve linear systems has been an open question for years. In this chapter, by relating growth factors and condition numbers with the run time variation for a floating point operation depending on precision, we suggest a practically sufficient condition in which LU with partial pivoting is superior to QR in MPIR in terms of run time given the same accuracy. We propose that LU with partial pivoting requires less run time than QR in MPIR given the same accuracy if $(2^{1/\beta} - 1)\cdot\log_2 \kappa(A) - \log_2\rho > 1$ when theoretically least sufficient precisions for LU and QR are applied, where ρ is a growth factor, $\kappa(A)$ is a condition number, and β is a run time variation depending on precision. We also explore this condition and its application to currently used customized static precision and configurable precision arithmetic units.

3-1. Growth factor in mixed precision iterative refinement

Success condition in MPIR depends on a matrix decomposition applied at step 3 in Algorithm II; we first look at how convergence rate can be affected by a matrix decomposition. We first explore success condition in MPIR in terms of 2-norm analysis.

Algorithm II. Iterative refinement :

A : a square matrix (n × n)
i = 0;

Step 1:	GEPP LU with partial pivoting	$O(n^3)$, precision ϵ_1
	$LU \times x^{(1)} = P \times b$	$O(n^2)$, precision ϵ_3
Step 2:	$r^{(i)} = A \times x^{(i)} - b$	$O(n^2)$, precision ϵ_2
Termination	? $\|\|r^{(i)}\|\|$ is small enough	
Step 3:	$LU \times z^{(i)} = P \times r^{(i)}$	$O(n^2)$, precision ϵ_3
Step 4:	$x^{(i+1)} = x^{(i)} - z^{(i)}$	O(n), precision ϵ_4

Termination ? $\|\|x^{(i+1)}\|\|$ is accurate enough

Go back to Step2 until $\|\|r^{(i)}\|\|$ or $\|\|x^{(i+1)}\|\|$ is accurate enough

Note.

GEPP: Gaussian Elimination with Partial Pivoting

P: Permutation matrix from GEPP

ϵ_i*: Precision applied at step i,*

r: Residual vector, x: Solution,

b: Right side vector in system equation (Ax = b)

Based on (2.15), the success condition in MPIR is :

$\|F\| \leq \|A^{-1}\| \|E\| / (1 - \|A^{-1}\| \|E\|) < 1$, where F is the convergence rate in the backward error, A is the matrix, and E is a backward error from direct method using either LU or QR. At this point, notice that E is a backward error to hold : $(A + E)(x + \delta x) = b$. (3.1) The E in (3.1) contains the information of not only the matrix factorization error but also the error to solve the linear system. We denote E_{LU} for the backward error for direct

method employing LU and E_{QR} for QR. Now, we explore E_{LU} and E_{QR}. This was previously explored in [61]. First, in LU, a linear system goes through the process :

$(L + \Delta L)\ \tilde{y} = b$ and $(U + \Delta U)\ \tilde{x} = \tilde{y}$; in QR, $(Q + \Delta Q)\ \tilde{y} = b$ and $(R + \Delta R)\ \tilde{x} = \tilde{y}$.

In order to find $\tilde{x}$, we need to solve : $(L + \Delta L)(U + \Delta U)\ \tilde{x} = b$. (3.2)

If we expand equation (3.2), $(L\ U + \Delta L\ U + L\ \Delta U + \Delta L\ \Delta U)\ \tilde{x} = b$. (3.3)

Comparing (3.1) to (3.3), $E_{LU} = \Delta L\ U + L\ \Delta U + \Delta L\ \Delta U$. (3.4)

Likewise, in QR, $E_{QR} = \Delta Q\ R + Q\ \Delta R + \Delta Q\ \Delta R$. (3.5)

The 2 norms of E_{LU} and E_{QR} can be represented as follow :

$$||E_{LU}||_2 \leq ||\Delta L||_2\ ||U||_2 + ||L||_2\ ||\Delta U||_2 + ||\Delta L||_2\ ||\Delta U||_2. \quad (3.6)$$

Since $A = LU$, $U = L^{-1}A$, and $L = A\ U^{-1}$, the equation (3.6) goes to (3.7):

$$||E_{LU}||_2 \leq ||L||_2 \cdot ||L^{-1}||_2 \cdot ||A||_2 \cdot ||\Delta L||_2/||L||_2 + ||U||_2 \cdot ||U^{-1}||_2 \cdot ||A||_2 \cdot ||\Delta U||_2/||U||_2 + (||\Delta L||_2/||L||_2) \cdot (||\Delta U||_2/||U||_2) \cdot ||L||_2 \cdot ||U||_2. \quad (3.7)$$

Since $||L||_2 \cdot ||U||_2 \leq ||L||_2 \cdot ||L^{-1}||_2 \cdot ||A||_2$ or $||L||_2 \cdot ||U||_2 \leq ||U||_2 \cdot ||U^{-1}||_2 \cdot ||A||_2$, the equation (3.7) can be represented as (3.8):

$$||E_{LU}||_2/||A||_2 \leq k_2(L) \cdot ||\Delta L||_2/||L||_2 + k_2(U) \cdot ||\Delta U||_2/||U||_2 + (||\Delta L||_2/||L||_2) \cdot (||\Delta U||_2/||U||_2) \cdot \min\{k_2(L), k_2(U)\}. \quad (3.8)$$

The equation (3.8) is also described in [61]. Likewise, in QR,

$$||E_{QR}||_2/||A||_2 \leq k_2(Q) \cdot ||\Delta Q||_2/||Q||_2 + k_2(R) \cdot ||\Delta R||_2/||R||_2 + (||\Delta Q||_2/||Q||_2) \cdot (||\Delta R||_2/||R||_2) \cdot \min\{k_2(Q), k_2(R)\}$$

$$= ||\Delta Q||_2/||Q||_2 + k_2(A) \cdot ||\Delta R||_2/||R||_2 + (||\Delta Q||_2/||Q||_2) \cdot (||\Delta R||_2/||R||_2).$$

Using the property (2.16),

$$||E_{QR}||_2/||A||_2 \leq ||\Delta Q||_2 + ||\Delta R||_2/||R||_2\ (k_2(A) + ||\Delta Q||_2). \quad (3.9)$$

Growth factors in triangular matrices from GEPP can make their condition numbers larger [61]. The least sufficient precision for matrix decomposition should consider this growth factor effects on accuracy in MPIR. The least sufficient precision for matrix decomposition in MPIR depends on the computing platform and matrix characteristics such as condition number and growth factor.

As for computing platform dependency, most customized static precision arithmetic units perform single and double precision arithmetic in hardware. Even though we are able to perform an extended precision arithmetic on most platforms, this requires extra software routines causing substantial overhead for performance. Therefore, it is desirable for users to utilize single and double precision in MPIR on the customized static precision arithmetic units. GPUs and multi-core processors are examples of common platforms that typically support single and double precision arithmetic in hardware. Some computing platforms employ configurable arithmetic units supporting arbitrary precision arithmetic in hardware. FPGAs represent an example that can support two arbitrary precisions in MPIR with configurable arithmetic units. GPUs and FPGAs are widely used in MPIR, since they are suitable for parallel computations [9, 22, 33, 41, 42, 45, 48].

As for matrix characteristic dependency, the condition number and growth factor affect the convergence in MPIR. The convergence rate in MPIR generally depends on the backward errors (i.e., ||E||/||A||) and condition numbers [36]. Given a matrix, the condition number is also given unless a pre-processing such as scaling is not performed, and the backward error depends on the growth factor, applied precision for matrix decomposition, the matrix decomposition method, and the right-side vector. When the

condition number is relatively large, the convergence may not be successful, since the relative error in the solution from a triangular system solver can exceed unity. If approximate solutions do not converge successfully given a condition number, we may employ a higher precision for matrix decomposition to reduce the backward error to make convergence successful. The condition number for linear system can be defined in various ways. A commonly used condition number for linear system is defined as: $\kappa(A) = ||A||\cdot||A^{-1}||$, where $||\cdot||$ is a subordinate norm [16, 35]. Skeel proposed another condition number for linear system solvers in [62]. Skeel's condition number is defined as: $\kappa(A)_{SK} = || \, |A^{-1}|\cdot|A| \, ||$. The relative backward errors depend not only on the precision, but also on which algorithm we apply for matrix decomposition. The stabilities for LU, Cholesky, and QR are discussed in [16, 63-65]. Either LU or QR can be used for any matrices but Cholesky factorization is used only for symmetric matrices. We discuss LU and QR in this dissertation. Even though either LU or QR can be employed in MPIR, most of the literature discusses LU with partial pivoting [9, 32, 41, 42], since LU with partial pivoting requires less computational work than QR and is practically stable [16, 63]. However, in some applications described in [60, 66, 67], large growth factors may appear in LU with partial pivoting and employing LU with partial pivoting in MPIR may not produce successful convergence. The growth factors become larger along with the matrix size [14]. A large growth factor may degrade the convergence rate severely in MPIR, since the backward error is generally magnified by the growth factor [6]. Even though QR requires twice the computational work of LU with partial pivoting, QR is unconditionally stable [57] -- its growth factor is relatively smaller than for LU. If the

growth factors from LU with partial pivoting are considerably larger than QR in some applications, it would be better to employ QR rather than LU with partial pivoting to make convergences successful. Since LU with partial pivoting is widely used, we refer to LU with partial pivoting as LU from this point forward for simplicity. Therefore, an applicable precision for matrix decomposition in MPIR considers the computing platform type, condition number, growth factor from LU (assuming that the growth factor in QR is negligible), and the type of matrix decomposition.

When programming MPIR, a user may wonder which matrix decomposition method could be better to be employed in MPIR given a matrix and a computing platform. A choice between LU and QR has been being an interesting question [16, 63] since Householder QR was proposed in 1958 [58]. Comparison between LU and QR for the direct method is discussed in [63]. In [63], the author mentions that LU can be widely used for linear system solvers since large growth factors from LU rarely happen in practice and the applications having large growth factors can be treated separately as unstable applications. In this dissertation, we seek a practically sufficient condition (i.e., (P 2) in section 3.2) in which LU is superior to QR in terms of runtime for MPIR given a matrix, a computing platform, and the same prescribed accuracy. To do so, we consider runtime variation for floating point operations depending on the precision. This is a sufficient condition since LU with partial pivoting could be superior to QR even though the condition is not satisfied. Even though this is not a strong condition (a necessary and sufficient condition), to our best knowledge, this is the first effort to quantify the relation that can describe a standard to tell when LU factorization is superior given a computing

platform and a matrix. The derivation of the sufficient condition is described in section 3-2. To make the sufficient condition practical, we apply the condition to currently used customized static precision arithmetic and configurable arithmetic units. Section 3-3 discusses how to apply the condition to the two types of arithmetic units. We also explore plots to visualize the condition according to different types of applications and computing platforms. The plots depend on matrix sizes, condition numbers, growth factors, and precision. The sufficient condition for SDIR has been tested using various matrices in MATLAB and the condition is verified in the test in section 3-4.

3-2. Preliminaries

In this section, we make some assumptions and discuss some preliminaries to derive the sufficient condition (P 3.2) in which LU is superior to QR in terms of run time given the same prescribed accuracy. The main difference in stability between LU and QR comes from the growth factor. If the growth factor impact on accuracy is negligible in a given matrix, we do not need to argue which matrix decomposition will be desirable, since LU is always superior to QR. Therefore, we consider applications in which growth factors can affect the solution accuracy to some extent. Also, we consider growth factors come only from LU: we assume that a growth factor in QR makes a negligible impact on accuracy in practice (i.e., the 2 norm of the upper triangular matrix in QR is the same as the 2 norm of the original matrix). We categorize input matrices according to their growth factors from LU. We denote input matrices as Medium Growth Factor (MGF) matrices if the growth factors follow $O(n^{1/2})$ (e.g., Gaussian random matrices [64]), as

Large Growth Factor (LGF) matrices if the growth factors follow $O(n)$ (e.g., applications shown in [67] such as Vandermonde-like matrices), and as eXtremely Large Growth Factor (XLGF) matrices if the growth factors follow $O(2^{n-1})$ (e.g., Volterra integral equations and two point boundary value problems [60, 66]). We derive the least sufficient precisions for LU and QR in MPIR first, then the required runtime, and finally the sufficient condition in which the runtime of MPIR employing LU is less than QR if LU and QR employ the least sufficient precisions for successful convergence.

To seek the least sufficient precision for matrix factorization in MPIR, we first explore a theoretical convergence rate given a condition number and growth factor. Based on Wilkinson's proof [36], the convergence rate for WIR is described in Lemma 3.1. MPIR performs step 3 using a lower precision and WIR employs the original precision for step 3. WIR employs a higher precision and MPIR employs the original precision for steps 2 and 4. The backward error (e.g., $||E||/||A||$) could be larger in MPIR than WIR, but the convergence analysis is the same. Hence, we apply the convergence rate for WIR to MPIR.

Lemma 3.1 (Wilkinson 1963 [36], (L 3.1)): If there exist no rounding-off errors in steps 2 and 4 in Algorithm II (practically, if the mantissa bit width for steps 2 and 4 is twice larger than the mantissa for matrix decomposition),

$$\Omega \leq (||E||/||A||)\, \kappa(A) / (1 - (||E||/||A||)\, \kappa(A)),$$

$$\Omega \leq \epsilon\, \kappa(A)_{SK} / (1 - \epsilon\, \kappa(A)_{SK})$$

where, Ω is a convergence rate given a matrix, $||\cdot||$ is a norm of a matrix, $|\cdot|$ is the matrix having absolute values of elements of a matrix, $||E||/||A||$ is a relative backward error from triangular system solver, ϵ is the smallest number which satisfies $|E| \leq \epsilon|A|$, $\kappa(A)$ is a normally used condition number, and $\kappa(A)_{SK}$ is a Skeel's condition number [62].

Proof 1 [36]: From (2.6),

$$||F|| = || A^{-1}E (I+A^{-1}E)^{-1}|| \leq || A^{-1}E ||\cdot||(I+A^{-1}E)^{-1}|| \leq || A^{-1}E || / (1 - ||A^{-1}E||)$$

$$\leq (||A^{-1}||\cdot||E||)/(1 - ||A^{-1}||\cdot||E||) = (||E||/||A||) \kappa(A)/(1 - (||E||/||A||) \kappa(A)).\ \text{Q. E. D.} \quad (3.10)$$

Proof 2: The proof can be performed using the relative error of the solution.

$$(A + E) (x + \delta x) = b,$$

$$Ax + Ex + A\delta x + E\delta x = b,$$

$$\delta x = -A^{-1}E(x+ \delta x), \quad (3.11)$$

$$|| \delta x || \leq || A^{-1}E || \, || x+ \delta x || \leq || A^{-1}E || \, (|| x || + || \delta x ||),$$

$$(1 - || A^{-1}E ||) \, || \delta x || \leq || A^{-1}E ||\cdot|| x ||,$$

$$|| \delta x ||/|| x || \leq || A^{-1}E || / (1 - || A^{-1}E ||)$$

$$\leq (||A^{-1}||\cdot||E||)/(1 - ||A^{-1}||\cdot||E||) = (||E||/||A||) \kappa(A)/(1 - (||E||/||A||) \kappa(A)).\ \text{Q. E. D.} \quad (3.12)$$

The (3.12) corresponds with (3.10). Hence, if the relative error in the solution from triangular system solver is less than unity, the approximate solutions can converge. The Skeel's condition number [62] can also express the condition for the successful convergence.

Proof 3 [62]: In (3.11),

$$\delta x = -A^{-1}E(x+\delta x)$$

$$|\delta x| \le |A^{-1}|\cdot|E|\,(|x| + |\delta x|) \le \epsilon\,|A^{-1}|\cdot|A|\,(|x| + |\delta x|)$$

$$|| \delta x || = || |\delta x| || \le \epsilon\,|| \,|A^{-1}|\cdot|\,A\,|| \cdot || |x| || + \epsilon\,|| \,|A^{-1}|\cdot|\,A\,|| \cdot || \,|\,\delta x\,|| $$

$$(1 - \epsilon\,|| \,|A^{-1}|\cdot|\,A\,||)\,|| \delta x || \le \epsilon\,|| \,|A^{-1}|\cdot|\,A\,|| \cdot || x ||$$

$$|| \delta x || / || x || \le \epsilon\,\kappa(A)_{SK} / (1 - \epsilon\,\kappa(A)_{SK}).$$ Q. E. D.

Backward error is usually small when the matrix decomposition has an excellent pre-conditioner [21]. The condition for successful convergence is represented as in corollary (C 3.1). A condition number represents the sensitivity of the solution respect to the changes in the problem data [62]. A practically defined condition number has importance in this paper, since the purpose of this paper provides a reasonable standard to compare the performances in MPIR between LU and QR given a prescribed accuracy and a computing platform. As one of practical condition numbers, a Skeel's condition number [62] is equal or less than the normally used condition number in terms of 1 and infinity norm as follows:

$|| \,|A^{-1}|\cdot|\,A\,||_\infty \le || \,|A^{-1}|\,||_\infty \cdot || \,|A\,|\,||_\infty = || A^{-1}||_\infty \cdot || A ||_\infty$. The condition numbers are exchangeable between the normally used condition number and Skeel's condition number, since the convergence formation is the same. Lemma 1 assumes that no rounding errors occur in residual calculation and updating solutions (i.e., Round off errors only come from the triangular system solvers). To eliminate rounding off errors, the required mantissa width for residual calculations and updating solutions should be

infinite in floating point arithmetic, which is not possible. Wilkinson and Moler suggested that doubled precision would produce sufficiently accurate results for residual calculation [36, 37] for WIR. This suggestion is still valid for MPIR and we show numeric test results for the suggestion in MPIR in section 4.

Corollary 3.1 (C 3.1) [36]:

Given a condition number, the condition for the successful convergence in MPIR is as follows: $||E||/||A|| < 0.5\kappa(A)^{-1}$ and $|E|/|A| < 0.5\kappa(A)_{SK}^{-1}$

Proof:

From (L 3.1), the convergence rate should be less than 1 for the successful convergence. Put $\Omega \leq (||E||/||A||)\,\kappa(A) / (1 - (||E||/||A||)\kappa(A)) < 1$. Arrange the inequality in terms of $||E||/||A||$ to obtain (C 3.1). The proof for Skeel condition numbers follows the same procedure. Q. E. D.

Now, we assume that the growth factor from QR is negligible in backward error and the growth factor from LU usually does not fully affect the accuracy. This assumption is represented as (A 3.1).

Assumption 3.1 (A 3.1):

$$||E_{QR}|| \leq ||A||\epsilon_L \text{ (or } |E_{QR}| \leq |A|\epsilon_L), \qquad \text{(A 3.1.1)}$$

$$||E_{LU}|| \leq \rho||A||\epsilon_L \text{ (or } |E_{LU}| \leq \rho|A|\epsilon_L). \qquad \text{(A 3.1.2)}$$

Therefore, to make the convergence successful,

$$\Omega_{QR} \leq (||E||/||A||)\cdot\kappa(A) / (1 - (||E||/||A||)\cdot\kappa(A)) \leq \kappa(A)\epsilon_L / (1 - \kappa(A)\epsilon_L) < 1. \quad \text{(A 3.1.3)}$$

From (L 3.1), $\Omega_{QR} \leq \kappa(A)_{SK}\, \epsilon_L / (1 - \kappa(A)_{SK}\, \epsilon_L) < 1$.

The backward can be measured by a posteriori backward errors as in Lemma 3.2 [62, 68, 69]. We simply refer the relative backward error to the backward error from this point forward in this paper.

Lemma 3.2 (L 3.2) [68, 69]: If an algorithm is backward stable, there exists a smallest E to satisfy

$||b - Ax_c|| / (||A||\cdot||x_c||) \leq ||E||/||A||$, where x_c is a computed solution from a triangular system solver.

Proof: $(A + E)\, x_c = Ax^* + Ex_c + A\delta x = b$. Therefore, $Ex_c = -A\delta x$.

$r = b - Ax_c = b - A(x^* + \delta x) = -A\delta x = Ex_c$,

$|| r || \leq ||E||\cdot||x_c||$,

$|| r || / (||A||\cdot||x_c||) \leq ||E||\cdot||x_c||/(||A||\cdot||x_c||) = ||E||/||A||$.

If $|E| \leq |A|\epsilon$,

$|r| \leq |E|\cdot|x_c| \leq |A|\cdot|x_c|\epsilon$, where $\epsilon = max_i\, (|r|_i/(|A|\cdot|x_c|)_i) \leq \epsilon_L$ in QR by (A 3.1.1) and in LU, $\epsilon = max_i\, (| r |_i/(|A|\cdot|x_c|)_i) \leq \rho\epsilon_L$.

Before seeking the appropriate lower precisions for LU and QR in MPIR, we assume two things for the run time for a floating point operation depending on precision [33] as in (A 3.2).

Assumption 3.2 (A 3.2): We assume that given a precision the run times for the four types of basic floating point operations (i.e., addition, subtraction, multiplication, and division) are equal. Even though a division requires substantially more run time than other floating point arithmetic, the number of divisions in our problem is relatively small. We also assume that the run time depends on precisions. Based on the assumptions, the run time per floating point operation is:

$$T(t) = \theta \cdot t^{\beta} \text{ and } \beta \geq 0 \quad \text{(A 3.2)}$$

where β represents the performance benefit as the mantissa bit width decreases, t is the mantissa bit width, and θ is a scaling factor to make (A 3.2) hold [18, 33] given a computing platform. For example, if the number of arithmetic units in a fixed area increases quadradically according to mantissa bit width decrease on FPGAs without clock rate variation, then $\beta = \text{'2'}$. If $\beta = \text{'0'}$, the run time is independent of precision.

Now, we seek appropriate lower precisions according to matrix factorizations. If the lower precision is too low, approximate solutions may not converge. If the lower precision is too high, it may cause performance loss in MPIR. The theoretically appropriate lower precisions are depicted in Property 3.1.

Property 3.1 (P 3.1): A theoretical least mantissa width for a lower precision for successful convergence in MPIR satisfies:

$t_{L\_QR} = minimum\ integer\ t > log_2\ \kappa(A)\ (t_{L\_QR} = log_2\ \kappa(A) + \xi_{QR},\ 0 < \xi_{QR} \leq 1)$.

The mantissa bit width for LU is:

$t_{L\_LU} = minimum\ integer\ t > log_2\ \rho\kappa(A)\ (t_{L\_LU} = log_2\ \rho\kappa(A) + \xi_{LU},\ 0 < \xi_{LU} \leq 1)$,

$t_{L\_LU} = log_2\ \rho\kappa(A) + \xi_{LU} = log_2\ \kappa(A) + log_2\ \rho + \xi_{LU} = t_{L\_QR} + log_2\ \rho + \xi_{LU} - \xi_{QR}$.

Proof: From (C 3.1) and (A 3.1), the theoretically appropriate lower precisions can be described as follows:

$||E_{QR}||/||A|| \leq \epsilon_{L\_QR} < 0.5\kappa(A)^{-1}$ for QR.

Hence, $\epsilon_{L\_QR} < 0.5\ \kappa(A)^{-1}$, $-log_2\ \epsilon_{L\_QR} > -log_2\ (0.5\ \kappa(A)^{-1})$ and take rounding mode for IEEE 754 [70] (i.e., $t_{L\_QR} + 1 = -log_2\ \epsilon_{L\_QR}$). $||E_{LU}||/||A|| \leq \rho\ \epsilon_{L\_LU} < 0.5\kappa(A)^{-1}$ for LU. Q. E. D.

$2n^3/3$ floating point operations are required for LU and $4n^3/3$ for QR. Refinement procedure (i.e., step 2 to 4 in Algorithm II) requires $O(n^2)$ while $O(n^3)$ for matrix decomposition. The number of iterations is generally much less than the size of matrix. Hence, the run time for refinement procedure can be negligible and we may consider the time it takes for matrix decomposition to estimate total run time for MPIR in practice unless the gap between the higher and lower precision is striking (e.g., when a user requires an extremely high precision accuracy). If a user requires an extremely high precision accuracy, the run time of refinement procedure cannot be negligible [18, 33],

since it is probable for the number of iterations to approach the matrix size. In Property 3.2, the precision is related with matrix characteristics to see the condition in that LU is superior to QR in MPIR in terms of run time, given a computing platform and a matrix.

Property 3.2 (P 3.2): If MPIR employs the theoretically least precisions based on (P 3.1) for QR and LU,

$$T_{LU} < T_{QR} \text{ if } (2^{1/\beta} - 1)\times log_2 \kappa(A) - log_2 \rho > 1 \quad \text{(P 3.2.1)}$$

$$(2^{1/\beta} - 1)\times log_2 \kappa(A) - log_2 \rho > 2^{1/\beta} \text{ if } T_{LU} < T_{QR} \quad \text{(P 3.2.2)}$$

where T_{QR} is the total run time for MPIR employing QR and T_{LU} for LU.

Proof: Based on (P 3.1),

If $(2^{1/\beta} - 1)\times log_2 \kappa(A) - log_2 \rho > 1$,

$$log_2 \rho < (2^{1/\beta} - 1)\,(t_{L\_QR} - \xi_{QR}) - 1 = (2^{1/\beta} - 1) log_2 \kappa(A) - 1,$$

$$t_{L\_QR} + log_2 \rho < 2^{1/\beta} t_{L\_QR} - \xi_{QR}(2^{1/\beta} - 1) - 1,$$

$$t_{L\_QR} + log_2 \rho + \xi_{LU} - \xi_{QR} < 2^{1/\beta} t_{L\_QR} - \xi_{QR}(2^{1/\beta}) + \xi_{LU} - 1,$$

Since $1/\beta > 0$, $2^{1/\beta} > 1$,

$$t_{L\_QR} + log_2 \rho + \xi_{LU} - \xi_{QR} < 2^{1/\beta} t_{L\_QR} - (\xi_{QR}(2^{1/\beta}) - \xi_{LU}) - 1 < 2^{1/\beta} t_{L\_QR}.$$

Since $T_{LU} = 2\alpha n^3/3\ t_{L\_LU}^{\beta} = 2\alpha n^3/3\ (t_{L\_QR} + log_2 \rho + \xi_{LU} - \xi_{QR})^{\beta}$ and $T_{QR} = 4\alpha n^3/3\ t_{L\_QR}^{\beta}$,

$$3/(2\alpha n^3)\ T_{LU} = t_{L\_LU}^{\beta} = (t_{L\_QR} + log_2 \rho + \xi_{LU} - \xi_{QR})^{\beta} < (2^{1/\beta} t_{L\_QR})^{\beta} = 2\, t_{L\_QR}^{\beta} =$$

$3/(2\alpha n^3)\ T_{QR}$. Q. E. D.

If $T_{LU} < T_{QR}$,

$$t_{L\_LU}^{\beta} < 2\, t_{L\_QR}^{\beta},$$

$log_2 \kappa(A) + log_2 \rho + \xi_{LU} < 2^{1/\beta} log_2 \kappa(A) + 2^{1/\beta} \xi_{QR}$,

$log_2 \rho < (2^{1/\beta} - 1) log_2 \kappa(A) + 2^{1/\beta} \xi_{QR} - \xi_{LU} < (2^{1/\beta} - 1) log_2 \kappa(A) + 2^{1/\beta}$. Q. E. D.

The former condition in (P 3.2.1) is more practically useful than the property (P 3.2.2), since users are interested in which matrix decomposition they will pick given a computing platform, a matrix, and the prescribed accuracy. Users may not have information about condition numbers and growth factors for given matrices before computation, but if users have information about rough condition numbers, the growth factor tendency and matrix sizes in advance, they are able to exploit the plots described in next section. The condition in (P 3.2.1) is a practically sufficient condition in which employing LU takes less execution time than QR in MPIR given a prescribed accuracy when MPIR employs the theoretically least precisions for LU and QR for successful convergence. For example, if the run time is independent of precision (i.e., $\beta = 0$), the condition always satisfies and we employ LU always. It is intuitively obvious that employing less computational work is always better if the run time is independent of precision. To apply this condition to more realistic situation, it can be modified as in (P 3.3).

Property 3.3 (P 3.3): Since a practical mantissa width can be less or equal than the theoretically least precision, a practical mantissa width for a lower precision in MPIR satisfies:

$t_{P\_QR} = t_{L\_QR} - \bar{\xi}_{QR}$, where $0 \leq \bar{\xi}_{QR} < t_{L\_QR}$,

$t_{P\_LU} = t_{L\_LU} - \bar{\xi}_{LU}$, where $0 \le \bar{\xi}_{LU} < t_{L\_LU}$.

Therefore, the practical run times are:

$$T_{P\_LU} = 2\alpha n^3/3\ t_{P\_LU}^{\beta} = 2\alpha n^3/3\ (t_{L\_QR} + log_2\rho + \xi_{LU} - \xi_{QR} - \bar{\xi}_{LU})^{\beta}$$

$$T_{P\_QR} = 4\alpha n^3/3\ t_{P\_QR}^{\beta}.$$

$T_{P\_QR} > T_{P\_LU}$ if $log_2\,\kappa(A)\,(2^{1/\beta} - 1) - log_2\rho > 2^{1/\beta}\,\bar{\xi}_{QR} - \bar{\xi}_{LU} + 1.$ (P 3.3.1)

$log_2\,\kappa(A)\,(2^{1/\beta} - 1) - log_2\rho > 2^{1/\beta}\,(\bar{\xi}_{QR} - 1) + \bar{\xi}_{LU}$ if $T_{P\_LU} < T_{P\_QR}$. (P 3.3.2)

Proof: If $log_2\,\kappa(A)\,(2^{1/\beta} - 1) - log_2\rho > 2^{1/\beta}\,\bar{\xi}_{QR} - \bar{\xi}_{LU} + 1$

$$2^{1/\beta}\,t_{L\_QR} > t_{L\_QR} - \xi_{QR} + \xi_{LU} + 2^{1/\beta}\,\xi_{QR} + log_2\rho + 2^{1/\beta}\,\bar{\xi}_{QR} - \bar{\xi}_{LU} - \xi_{LU} + 1,$$

$$2^{1/\beta}\,t_{L\_QR} > t_{L\_LU} + 2^{1/\beta}\,\xi_{QR} + 2^{1/\beta}\,\bar{\xi}_{QR} - \bar{\xi}_{LU} - \xi_{LU} + 1 > t_{L\_LU} + 2^{1/\beta}\,\bar{\xi}_{QR} - \bar{\xi}_{LU},$$

$$2^{1/\beta}\,t_{P\_QR} > t_{P\_LU},$$

$T_{P\_QR} > T_{P\_LU}$. Q. E. D.

If $T_{P\_LU} < T_{P\_QR}$, $t_{L\_QR}\,(1 - 2^{1/\beta}) + log_2\rho < -\xi_{LU} + \xi_{QR} + \bar{\xi}_{LU} - 2^{1/\beta}\,\bar{\xi}_{QR}$, based on (P 3.1),

$log_2\,\kappa(A)\,(2^{1/\beta} - 1) - log_2\rho > 2^{1/\beta}\,(\bar{\xi}_{QR} - \xi_{QR}) - (\bar{\xi}_{LU} - \xi_{LU}) > 2^{1/\beta}\,\bar{\xi}_{QR} - \bar{\xi}_{LU} - 2^{1/\beta}$. Q. E. D.

To make the sufficient condition $log_2\,\kappa(A)\,(2^{1/\beta} - 1) - log_2\rho > 1$ valid in practice, the condition $\bar{\xi}_{LU} \ge 2^{1/\beta}\,\bar{\xi}_{QR}$ should be satisfied based on (P 3.3.1). It is less probable that the sufficient condition can be applied to real applications if either β is small or the gap in the mantissa width between a theoretically defined least precision and a practical least precision is smaller or equal in LU compared to QR. Practically, in current computing platform, β is in the range in [0.85, 2]. Hence, even in a computing platform possessing smallest β, the sufficient condition can be applied as long as $\bar{\xi}_{LU} \ge 2.3\,\bar{\xi}_{QR}$. That case is

reasonable in practice since the growth factor from LU does not fully affect the backward error in practice; the mantissa width gab between the theoretically least precision and a practically applicable least precision would be much larger in LU compared to QR.

The property (P 3.2) depends on an applied computing platform, since β represents the run time variation depending on precision and it may have different values according to an applied computing platform. Hence, we apply this condition to widely used computing platforms such as multi-cores, GPUs, and FPGAs. We first categorize the accelerators into two types as follows.

Accelerator Type 1 (AT 1): They have single and double precision arithmetic units in hardware and single precision computation is twice faster than double precision computation -- NVIDIA GTX 480 GPU or multi-cores.

Accelerator Type 2 (AT 2): They employ arbitrary precision ALUs in hardware and the computation speed increases approximately quadradically according to mantissa bit width decrease -- FPGAs.

β can be estimated based on (A 3.2). If the run time for a single precision floating point arithmetic is a half of the run time for double precision, the equality of $2T(t_{single}=23) = T(t_{double}=53)$ makes β approximately 0.85. The number of arithmetic units for multipliers in MPIRs in a fixed area can be increased quadradically on FPGAs according to mantissa bit width decrease [9]. We assume that clock speed variation is not significant when we increase the number of arithmetic units on FPGAs (i.e., the clock

speed might be improved by employing the smaller precision arithmetic units, but the place and route to connect multiple arithmetic units would be more complicated, which may degrade the speed). Hence, we assume the βs for (AT 1) and (AT 2) as in (A 3.3).

Assumption 3.3 (A 3.3): $\beta = 0.85$ for (AT 1) type (e.g., GPUs or multi-cores) and 2 for (AT 2) type (e.g., FPGAs).

Based on (A 3.3), we derive the properties (P 3.3 - 4) to draw the plots in the subsequent section.

Property 3.3 (P 3.3):

$T_{LU} < T_{QR}$ if $log_2 \rho > (2^{1.177} - 1) \times log_2 \kappa(A)$ for (AT 1) type. (P 3.3.1)

$log_2 \rho > (2^{0.5} - 1) \times log_2 \kappa(A)$ for (AT 2) type. (P 3.3.2)

Proof: It is derived from (A 3.3) and (P 3.2). Notice that LU is better than QR in terms of run time for a computing platform having small βs.

Property 3.4 (P 3.4): Allowable condition numbers for LU and QR

(AT 1) computing platform :

$\kappa(A) < 2^{23}$ for single precision QR, $\kappa(A) < 2^{23}/\rho$ for single precision LU. (P 3.4.1)

$\kappa(A) < 2^{52}$ for double precision QR, $\kappa(A) < 2^{52}/\rho$ for double precision LU. (P 3.4.2)

(AT 2) computing platform :

$\kappa(A) < 2^{ta}/\rho$ for arbitrary precision LU and $\kappa(A) < 2^{ta}$ for arbitrary precision QR, where *ta* is an arbitrary precision. (P 3.4.3)

Proof: For QR, $t_{L\_QR} > log_2 \kappa(A)$. For single precision, $23 > log_2 \kappa(A)$. Therefore, $\kappa(A) < 2^{23}$. For LU, $t_{L\_LU} > log_2 \rho + log_2 \kappa(A)$. For single precision, $23 - log_2 \rho > log_2 \kappa(A)$. Therefore, $\kappa(A) < 2^{23}/\rho$. The proofs for double precision and arbitrary precision go through the same procedure as the proof for single precision. Q. E. D.

3-3. The sufficient condition for computing platforms

In this section, we visualize the sufficient condition (P 3.2) for (AT 1) and (AT 2) computing platforms. Figure 5 represents the property (P 3.2). The *x* axis represents the log_2 based condition numbers and the *y* axis represents the log_2 based growth factors. Each arrow points out the area in which the sufficient condition of (P 3.2) is satisfied for (AT 1) and (AT 2). The satisfied area is much larger in (AT 1) than (AT 2), since the β is smaller. Figure 5 does not show the required mantissa to make convergence successful. In order to seek a required mantissa width, it is necessary to visit (P 3.1). Based on (P 3.1), the value on the *x* axis approximately represents required mantissa width for QR and the $(x + y)$ value approximately represents required mantissa width for LU. Growth factors generally become larger along with the matrix sizes. We explore applications having $\rho = O(n^{0.5})$ [64] (i.e., MGF), $O(n)$ [66, 67] (i.e., LGF), and $O(2^{2n-1})$ [60, 66] (i.e., XLGF) and we take the maximum growth factors for the applications such as $\rho = n^{0.5}$ for MGF, *n* for LGF, and 2^{2n-1} for XLGF matrices to satisfy the sufficient condition (P 3.2) regarding to the growth factors.

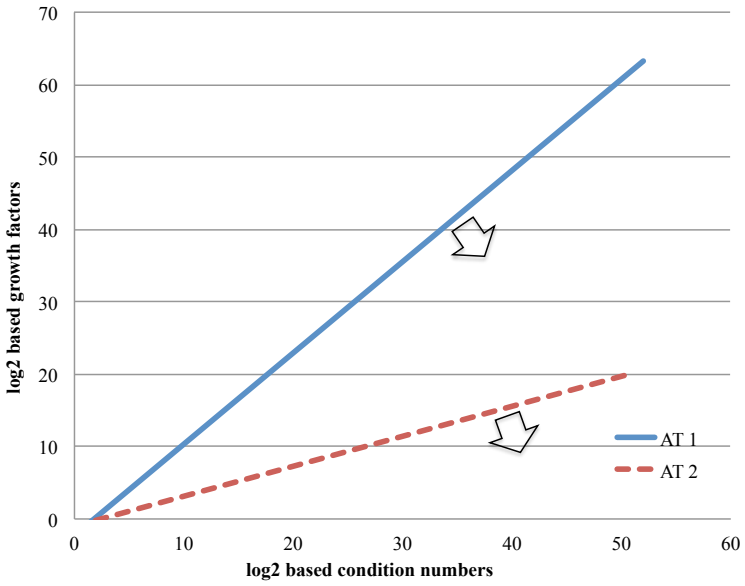

Figure 5. Sufficient conditions for $T_{LU} < T_{QR}$

We first apply such applications to (AT 1). Figure 6 shows a plot representing the sufficient condition of (P 3.2) for (AT 1) according to the three types of applications. The x axis represents the log_2 based condition numbers and the y axis the log_2 based matrix sizes. The plot is in the ranges, $1 \leq \kappa(A) \leq 2^{51}$ and $1 \leq n \leq 2^{51}$. The plot considers $\beta = 0.85$. A computing platform (AT 1) is able to utilize only single and double precision in MPIR. Therefore, we consider possible four choices in the plot in SDIR such as LU with single, QR with single, LU with double, and QR with double precision. Run time for LU with double precision is the same as the run time for QR with single precision in SDIR by (A 3.3), in which case we take QR with single precision. The arrows indicate the areas in which the sufficient conditions (P 3.2) are satisfied given an application.

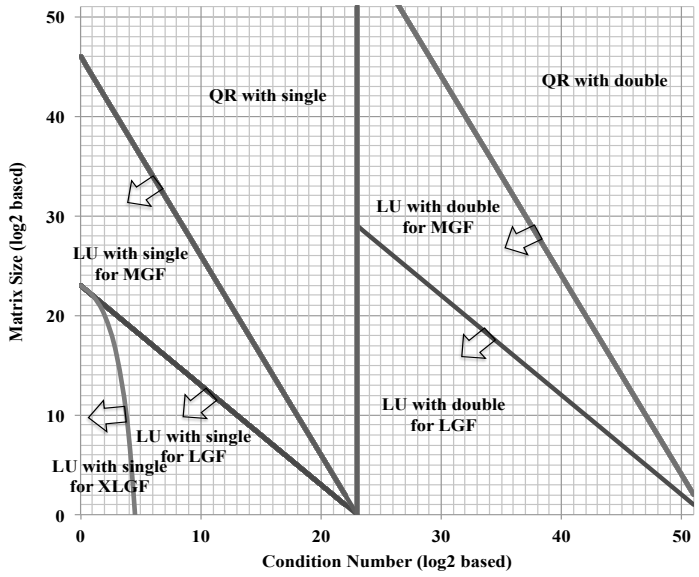

Figure 6. Decision plot for single and double precision iterative refinement

For example, given an MGF matrix, even though the intercept point by a condition number and a matrix size is in the area for LU with single for LGF, we can still apply LU with single for MGF matrices. To see if the plot follows (P 3.2), consider a nearly boundary point for LGF matrices at which condition number = $2^{22.9}$ and matrix size = 2^{29}. The sufficient condition of (P 3.3) does not hold (e.g., $(2^{1.177} - 1) \times 22.9 = 28.9 - 29 < 1$), but the inequality is practically equal; LU with double precision is practically as a good method as QR with single precision at the intercept point.

The sufficient condition (P 3.2) can be applied to (AT 2) as shown in Figure 7. Each arrow in Figure 7 represents the sufficient condition (P 3.2) according to the three types of applications. The *x* axis represents log_2 based condition numbers and the *y* axis represents the log_2 based matrix sizes.

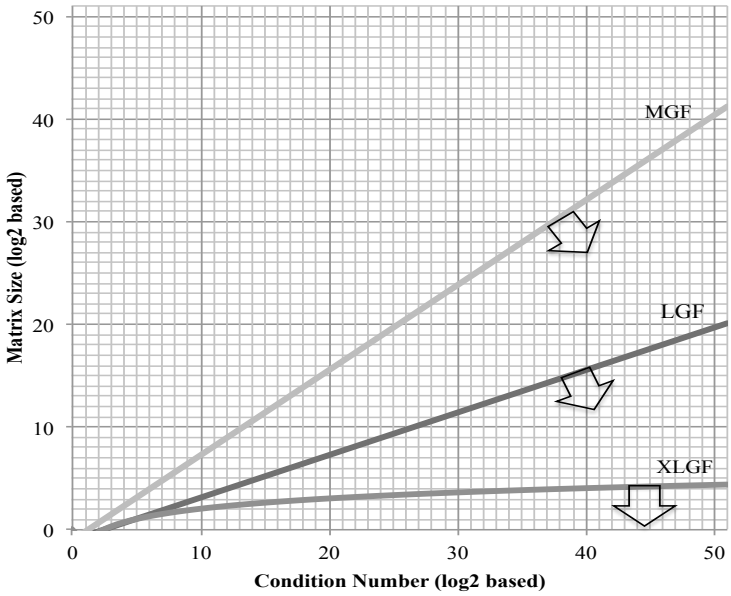

Figure 7. Decision plot for extended mixed iterative refinement

Required mantissa width for QR can be approximately represented by the values on the *x* axis. The required mantissa width for LU can be approximately represented by *(x+y/2)* for MGF, *(x+y)* for LGF, and $(x+2^{y-1})$ for XLGF matrices. For example, suppose you have a Gaussian random matrix application (i.e., MGF), when the condition number is 2^{20} and the matrix size is 2^{10}, the sufficient condition is satisfied and employing LU with 25 bits for the mantissa in XMIR is recommended (Strictly speaking, the required mantissa width is 26 bits by (P 3.1), but usually 1 bit difference is generally acceptable in practice (refer to P 3.2.1)).

To apply the plots for real applications, users should know rough condition numbers and growth factor tendency for the matrix. For example, if an application

follows growth factor between MGF and LGF, then users may choose LGF for safety. Growth factor tendency and condition number according to applications can be sought either theoretically or empirically [60, 66, 67]. If a user is not sure about growth factor for the application, we recommend to apply MGF since the matrix elements during LU factorization process follow normal distribution with high probabilities [64]. Another literature also states that a factor empirically found from various kind matrices for successful convergence (i.e., γ in [27] behaves a very similar way with the growth factor defined in this paper) is bounded the square root of a matrix size like MGF cases. Based on Figure 6 and 7, it is obvious that *LU is desirable in most cases for applications possessing small growth factors and βs and QR is desirable for applications possessing large growth factors and βs.*

3-4. Case study

Three types of assumptions such as (A 3.1-3.3) are made based on some previous literature's suggestions [9, 36, 53] to derive the sufficient condition (P 3.2). The (A 3.1) is related to numeric behavior and (A 3.2-3.3) are related to computing platforms. We examine (A 3.1) to see if the assumption for the backward errors is practically reasonable according to MGF, LGF and XLGF cases. To do so, we measure the backward errors using (L 3.2) and employ single precision for matrix decomposition and double precision for residual calculation so that round-off error from step 2 is relatively small compared to the backward error from step 3. In chapter 4, we explore a least sufficient precision empirically for steps 2 and 4 to make Lemma 1 legit in practice; the round-off errors from

double precision arithmetic in step 2 are negligible compared to the backward error from single precision arithmetic in step 3 empirically.

We measures the backward errors in MATLAB for 10 matrices each for MGF, LGF and XLGF when n = 1K. We choose Gaussian random matrices for MGF applications, a Vandermonde-like matrices (i.e., (2.2) in [67]) for LGF cases, and a boundary value problem for XLGF cases (i.e., A boundary value problem (second example in [60])). We generate the Gaussian random matrices using *randn(n)* command in MATLAB. The measured backward errors are divided by the precision applied for matrix decompositions (i.e. single precision, 2^{-24}) to observe clearly if the backward errors from QR are bounded by unity as in (A 3.1). Table III shows the measured average backward error, maximum backward error, and their variance for MGF cases, Table IV for LGF cases, and Table V for XLGF cases. In each row in the tables, first sub-row represents 2 norm backward errors and second sub-row component-wise backward errors. The corresponding condition numbers are $||A||_2 \cdot ||A^{-1}||_2$ for 2 norm condition number and $|| \, |A^{-1}| \cdot |A| \, ||_2$ for Skeel's condition number. Based on Table III, IV and V, the backward errors divided by the precision applied for QR are bounded the order of unity (e.g., less than 10) except component-wise backward errors for XLGF. Norm-wise backward errors from QR seem to grow $O(n^{1/3})$ in MGF and LGF cases; even in large matrices, the effects of matrix sizes are not significant in practice.

TABLE III. BACKWARD ERRORS FOR MGF MATRICES

Matrix size	64	256	1024
$\|\|E_{QR}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	2.3 / 2.7 / 0.07	3.4 / 3.6 / 0.02	5.1 / 5.5 / 0.06
$\|E_{QR}\|/\ \epsilon_s\|A\|$	2.4 / 2.9 / 0.14	2.4 / 3.1 / 0.23	2.1 / 2.7 / 0.10
$\|\|E_{LU}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	3.5 / 4.9 / 0.47	16.4 / 19.4 / 4.69	88.8 / 108.2 / 78.8
$\|E_{LU}\|/\ \epsilon_s\|A\|$	4.2 / 6.3 / 1.67	11.4 / 14.0 / 4.13	36.5 / 47.0 / 31.7
ρ	3.8 / 5.4 / 0.60	8.3 / 11.3 / 1.79	18.1 / 22.2 / 5.3
$\|\|E_{LU}\|\|_2/\ \rho\epsilon_s\|\|A\|\|_2$	0.9 / 1.2 / 0.04	2.0 / 2.9 / 0.23	5.0 / 6.4 / 0.77
$\|E_{LU}\|/\ \rho\epsilon_s\|A\|$	1.2 / 1.9 / 0.19	1.4 / 2.1 / 0.15	2.0 / 2.8 / 0.22

TABLE IV. BACKWARD ERRORS FOR LGF MATRICES

Matrix size	64	256	1024
$\|\|E_{QR}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	4.8 / 5.2 / 0.09	6.9 / 7.8 / 0.24	11.7 / 12.6 / 0.19
$\|E_{QR}\|/\ \epsilon_s\|A\|$	3.1 / 4.3 / 0.27	2.6 / 3.2 / 0.19	2.6 / 2.9 / 0.09
$\|\|E_{LU}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	9.8 / 12.7 / 2.2	30.5 / 34.1 / 5.70	112.8 / 130.0 / 110.4
$\|E_{LU}\|/\ \epsilon_s\|A\|$	6.9 / 8.5 / 1.5	13.3 / 15.6 / 2.67	28.9 / 38.6 / 28.4
ρ	32	128	512
$\|\|E_{LU}\|\|_2/\ \rho\epsilon_s\|\|A\|\|_2$	0.3 / 0.4 / 0.002	0.2 / 0.3 / 3.5×10^{-4}	0.2 / 0.3 / 4.2×10^{-4}
$\|E_{LU}\|/\ \rho\epsilon_s\|A\|$	0.2 / 0.3 / 0.002	0.10 / 0.12 / 1.6×10^{-4}	0.06 / 0.08 / 1.1×10^{-4}

TABLE V. BACKWARD ERRORS FOR XLGF MATRICES

Matrix size	16 (L = 10)	31 (L = 20)	
$\|\|E_{QR}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	0.6 / 0.9 / 0.02	0.5 / 0.6 / 0.008	
$\|E_{QR}\|/\ \epsilon_s\|A\|$	7.4 / 21.6 / 30.8	12.1 / 28.5 / 59.5	
$\|\|E_{LU}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	452.9 / 1057.4 / 7.7×10^4	1.2×10^6 / 1.5×10^6 / 8.8×10^{10}	
$\|E_{LU}\|/\ \epsilon_s\|A\|$	1331.1 / 3688.2 / 9.4×10^5	5.3×10^6 / 9.2×10^6 / 5.3×10^{12}	
ρ	3.6×10^3	1.2×10^8	
$\|\|E_{LU}\|\|_2/\rho\epsilon_s\|\|A\|\|_2$	0.12 / 0.29 / 0.006	0.010 / 0.013 / 6.2×10^{-6}	
$\|E_{LU}\|/\ \rho\epsilon_s\|A\|$	0.37 / 1.01 / 0.071	0.04 / 0.08 / 3.8×10^{-4}	
Matrix size	64 (L = 41)	64 (L = 21)	64 (L = 18)
$\|\|E_{QR}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	0.6 / 0.7 / 0.007	0.7 / 0.9 / 0.014	0.7 / 0.9 / 0.008
$\|E_{QR}\|/\ \epsilon_s\|A\|$	26.4 / 45.7 / 125.3	15.7 / 28.6 / 53.0	24.6 / 70.0 / 357.9
$\|\|E_{LU}\|\|_2/\ \epsilon_s\|\|A\|\|_2$	$1\times10^6/2\times10^6/2\times10^{11}$	$1\times10^6/2\times10^6/1\times10^{11}$	$3\times10^5/6\times10^5/4\times10^{10}$
$\|E_{LU}\|/\ \epsilon_s\|A\|$	$2\times10^7/2\times10\ /1\times10^3$	$1\times10^7/1\times10^7/1\times10^{13}$	$1\times10^6/2\times10^6/9\times10^{11}$
ρ	3.4×10^{17}	3.4×10^8	1×10^7
$\|\|E_{LU}\|\|_2/\rho\epsilon_s\|\|A\|\|_2$	$3\times10^{-12}/7\times10^{-12}/2\times10^{-24}$	$0.006/0.009/2\times10^{-6}$	$0.03/0.05/3\times10^{-4}$
$\|E_{LU}\|/\ \rho\epsilon_s\|A\|$	$5\times10^{-11}/5\times10^{-11}/9\times10^{-33}$	0.04 / 0.06 / 3×10^{-4}	0.13 / 0.23 / 0.008

In XLGF matrices, the norm-wise backward errors divided by the precision are bounded by unity. Therefore, we recommend using 2 norm condition numbers instead of using Skeel's condition numbers in XLGF cases for the sufficient condition of (P 3.2). The backward errors from LU divided by the applied precision times growth factor is bounded unity in LGF and XLGF cases. In MGF cases, the backward error growth rate according to matrix sizes seems to be bounded to $O(n^{1/2})$ (This corresponds to [64]).

Growth factors obviously can affect the backward errors in floating point arithmetic, since the smaller elements should be shifted right according to the growth factor before the addition (or subtraction). However, if the growth factors are too large compared to the elements of matrix, the large growth factors do not fully affect the accuracy in the solutions. We check the backward errors when the growth factors are larger than the reciprocal of the double precision times the largest absolute elements of matrices. The right side vector is set to all '1's. Table VI shows the norm-wise and component-wise backward errors obtained from single and double precisions. Notice that the backward errors are not divided by the applied precisions applied unlike previous tables.

TABLE VI. BACKWARD ERRORS FOR XLGF FROM SINGLE AND DOUBLE PRECISION

Matrix size	64 (L = 40)	256 (L = 40)	1K (L = 40)
$\|\|E_{LU}\|\|_2/\|\|A\|\|_2$ (single)	0.039	0.030	0.039
$\|E_{LU}\|/\|A\|$ (single)	1.0	1.0	1.0
$\|\|E_{LU}\|\|_2/\|\|A\|\|_2$ (double)	0.047	0.029	0.040
$\|E_{LU}\|/\|A\|$ (double)	0.94	0.92	0.81
ρ	$2^{56.64}$	$2^{55.13}$	$2^{55.10}$

Based on the table, if the growth factor is extremely large, we do not necessarily employ a higher precision, since the backward errors are not improved. Even though this statement is not strikingly new to readers, but we believe it is worthy to mention this since this seems to be opposite of a common belief that a higher precision "guarantees" a smaller backward error. The common belief might be true if the mantissa width for a higher precision is larger than the growth factor divided by the maximum absolute element size of a matrix. Therefore, we define a practical growth factor which can *actually* affect the backward error: *If* $\rho > max(|A|)/\epsilon$, $\rho_P = 1/\epsilon$; *else* $\rho_P = \rho$.

We test the plots in Figure 6 to see if the sufficient conditions are satisfied. Since it is important how closely the sufficient conditions fit in practice, we examine near the boundary points for SDIR. First, we examine XLGF cases. Let us take the second example in [60] (i.e., $n = 61$, $L = 40$, $k = 1$, and $\kappa(A)_2 = 88$). In the plot, it corresponds $log_2(n) \approx 5.9$, $log_2(\kappa(A)_2) \approx 6.5$, and $log_2(\kappa(A)_{SK\_2}) \approx 5.4$. Even though it fails the sufficient condition to employ LU with single precision, the successful convergence occurs in the experiment using LU with single precision. Therefore, we expect the sufficient conditions are satisfied inside the region of LU with single for (AT 1). To increase the condition numbers, we change n and L to $n = 91$ and $L = 60$ [60]; $log_2(n) \approx 6.5$, $log_2(\kappa(A)_2) \approx 7.1$ and $log_2(\kappa(A)_{SK\_2}) \approx 34.5$. Notice that Skeel's condition number is increased significantly, but the 2 norm condition number is not. In this case, only 86 out of 1000 matrices show the successful convergence (i.e. around 9 % success). The difference on the x axis is around 3 points in terms of 2 norm condition numbers.

For LGF cases, we test a 256 × 256 matrix of (2.2) in [67] with 1000 random right side vectors. The matrix has $log_2(n) = 8$, $log_2(\kappa(A)_2) \approx 0.5$, and $log_2(\kappa(A)_{SK\_2}) \approx 7.7$. If the region is in sufficient condition area, all the test cases show the successful convergences using LU with single precision. When we increase the matrix size so that $log_2(n) = 12$, $log_2(\kappa(A)_2) \approx 0.5$, and $log_2(\kappa(A)_{SK\_2}) \approx 11.7$, employing LU with single precision still shows 100 % successful convergence.

For MGF cases, we intentionally increase condition numbers by subtracting one of real eigenvalues with adding single precision perturbation on diagonal elements from the original matrices. Whenever a convergence fails in MPIR, we examine the Skeel's and 2 norm condition numbers and the growth factors. We tested 500 256 × 256 Gaussian random matrices in SDIR and right side vectors are generated with normal distribution too. On this experiment using the intentionally manipulated Gaussian random matrices, the average growth factor was 8.3 and one of 500 matrices was not able to converge. The log_2 based condition numbers are *34.1* for 2 norm and *36.4* for Skeel's condition number. The average number of iterations is 2.21 for 499 matrices having successful convergences.

3-5. Summary of chapter

MPIR using LU is widely used, but in some applications a large growth factor from LU may cause convergence to make failure. Householder QR is unconditionally numerically stable but it requires twice more computational work than LU. In this chapter, we describe a practically sufficient condition quantitatively to choose LU by

linking a run time depending on precision with system matrix characteristics. The two plots derived by the sufficient condition (P 3.2) may provide users a way to implement MPIR effectively according to their own computing platforms and the plots can be revised by changing the value of β according to diverse computing platforms. One of the plots (i.e., SDIR) is tested and the sufficient condition is satisfied for applications having various growth factors. We believe the sufficient condition may provide numerical analysts and computational scientists a way to conceive the linking between the current computation technologies with matrix theories. The sufficient condition also ameliorates the curiosity which matrix decomposition is more practically useful for dense linear system applications between LU and QR given a computing platform and a matrix.

Chapter 4

Average case for mixed precision iterative refinement

Numeric analyses for algorithms usually provide error bounds for worst cases, but the error bounds for the worst cases seem never happen in practice. It would be beneficial to analyze numeric behavior statistically to predict plausible results in real applications [64, 71]. Trefthen explored the average cases for residuals and backward errors for direct methods employing LU and QR according to matrix sizes for various random matrices [64]. He suggested that the relative error growth for residual is O(n) for LU and $O(n^{1/2})$ for QR.

In this chapter, we seek the average case of convergence rates for iterative refinements employing LU and QR. First, we explore average cases for a practically sufficient condition for a higher precision in which does not affect the convergence rate in practice in section 4-2. Second, we explore the average case for the dependency of convergence rate on a lower precision and a matrix size in section 4-3 and 4-4. Based on the results from section 4-3 and 4-4, we seek average case for convergence rates for iterative refinements employing LU and QR. Finally, based on the average convergence rates, we seek a condition for a practical lower precision to produce high performance in iterative refinements employing LU and QR. Since the infinity norms of condition numbers are widely used in real computation or theoretical work, we seek the average convergence rates depending on applied precisions and infinity condition numbers using large Gaussian Random Matrices (GRMs).

GRMs are used for some applications [12] and have a special property of the condition number distribution according to matrix sizes. Random matrix theories and applications are discussed in [72-75]. To generate random matrices, random number generators are required. If you want to use them for accelerators such as GPUs and FPGAs, it would be beneficial to generate random numbers inside accelerators to reduce run time for data transfer from host to accelerators. For example, we might use Hardware Accelerated Random Number Generators (HASPRNG) [76-78] to generate random matrices. HASPRNG represents an hardware implementation of SPRNG [79] for integer uniformly distributed random number generation. Random number generators depending on emerging architectures are compared each other in [80].

4-1. A practical condition for a higher precision

Based on numeric analysis performed in chapter 2, the infinite precision is theoretically required for the higher precision to keep the best convergence rates. Since the infinite precision is not possible to be implemented, we seek practical condition for the original precision, which is sufficient not to affect the convergence rate determined in step 3. To seek this condition, we fix the matrix size and lower precision and vary the original precision to observe the convergence rates according to the variation of original precision. The convergence rates are measured based on the difference from the first two residuals (i.e., residuals at 0^{th} and 1^{st} iteration in Algorithm II). Based on Moler's analysis, if the condition, $\epsilon_L >> \epsilon_A$ is satisfied, the convergence rate is independent of original precision. If there is some constant η so that it makes convergence rate

independent of higher precision in practice when $t_L \leq \eta t_A$, we want to explore η empirically.

We experiment MPIRs employing GEPP with 10,000 64×64 matrices each for t_A = 16, 20, 24, 28, 32, 35 and 52 bits when t_L = 16 bits. Figure 8 shows convergence rates according to different original precisions. In the figure, the red dots represent the convergence rates for t_A = 16, blue dots for t_A = 20, and green dots for t_A = 32. Employing a higher precision than t_A = 32 does not improve convergence rate in practice in this experiment. Therefore, we set $\eta = 1/2$ to make original precision independent for convergence rates. Interestingly, when condition numbers are large, the convergence rates are similar each other.

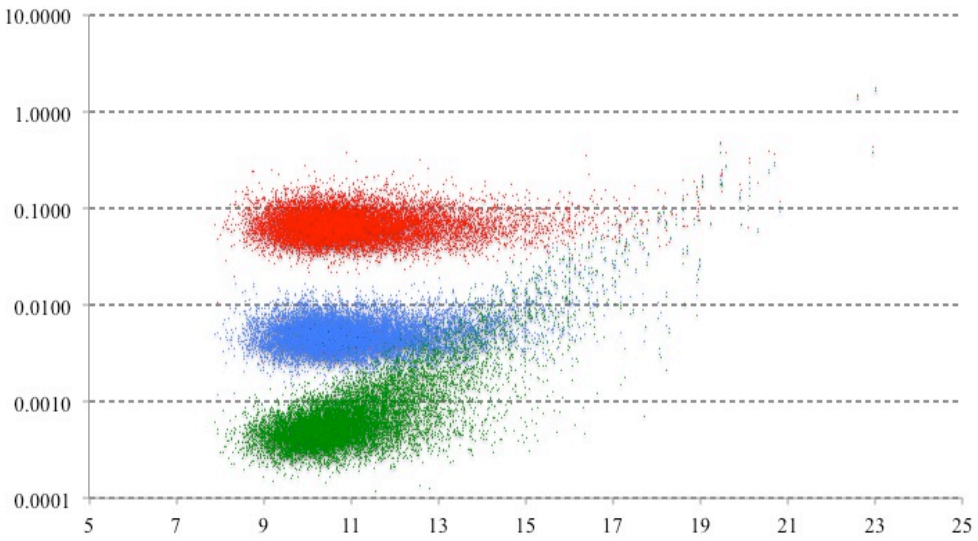

Figure 8. Dependecy on higher precison

Property 4.1 (P 4.1):

In average cases, if $t_A/t_L \geq 2$, the convergence rates in MPIRs employing GEPP are rarely affected by the precision for steps 2 and 4.

4-2. Dependency of convergence rate on a lower precision

In this experiment, we employ double precision for steps 2 and 4 for n = 64 and vary lower precisions. We experiment MPIRs employing GEPP with 10,000 64×64 matrices each for t_L = 16, 21, and 26 when t_A = 52 bits. Figure 9 shows the convergence rates according to various lower precisions. Based on the figure, the convergence rates have a linear relation according to lower precisions.

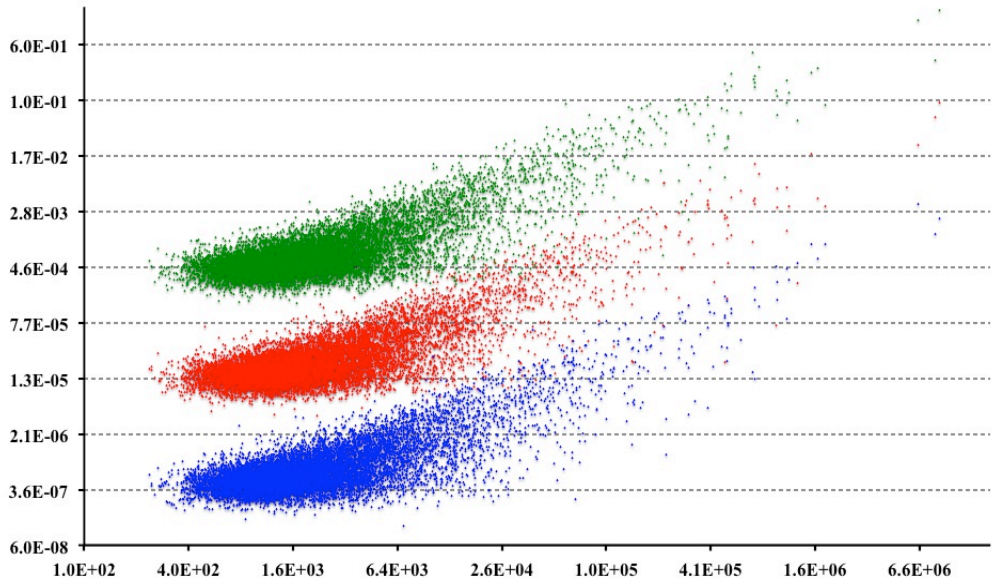

Figure 9. Dependecy on lower precison

Property 4.2 (P 4.2): In average cases, the convergence rates for iterative refinements employing GEPP are improved linearly according to lower precisions.

4-3. Dependency of convergence rate on matrix sizes

Convergence rates for SDIR employing GEPP and QR have been explored with 1000 GRMs generated by GSL library [81] each for eight different values of n, ranging from 64 to 8K at power of two increments. MAGMA v0.1 library [44] is employed to perform matrix decompositions on the NVIDIA Tesla C1060 and the LAPACK v3.6 library to perform the refinement procedures on host.

To seek average converge rates depending on matrix size, we employ double precision for steps 2 and 4, a lower precision as single precision (i.e., SDIR), and vary matrix sizes. Figure 10 shows the convergence rates for SDIRs employing LU (top in the figure) and QR (bottom) when n = 64 and 8K. Blue dots represent the convergence rates when n = 64, and red dots when n = 8K. Based on the figure, tracking down the worst convergence points according to condition numbers, the straight line seems to be constructed in LU, but not in QR. Hence, the convergence rates for SDIR employing LU are almost independent of matrix sizes according to infinity norm based condition numbers, but not for QR. Interestingly, increasing matrix sizes, the convergence rates for SDIRs employing QR corresponding to condition numbers become improved. The behavior would be different if we measure the convergence rates according to 2-norm based or Skeel's condition numbers [62].

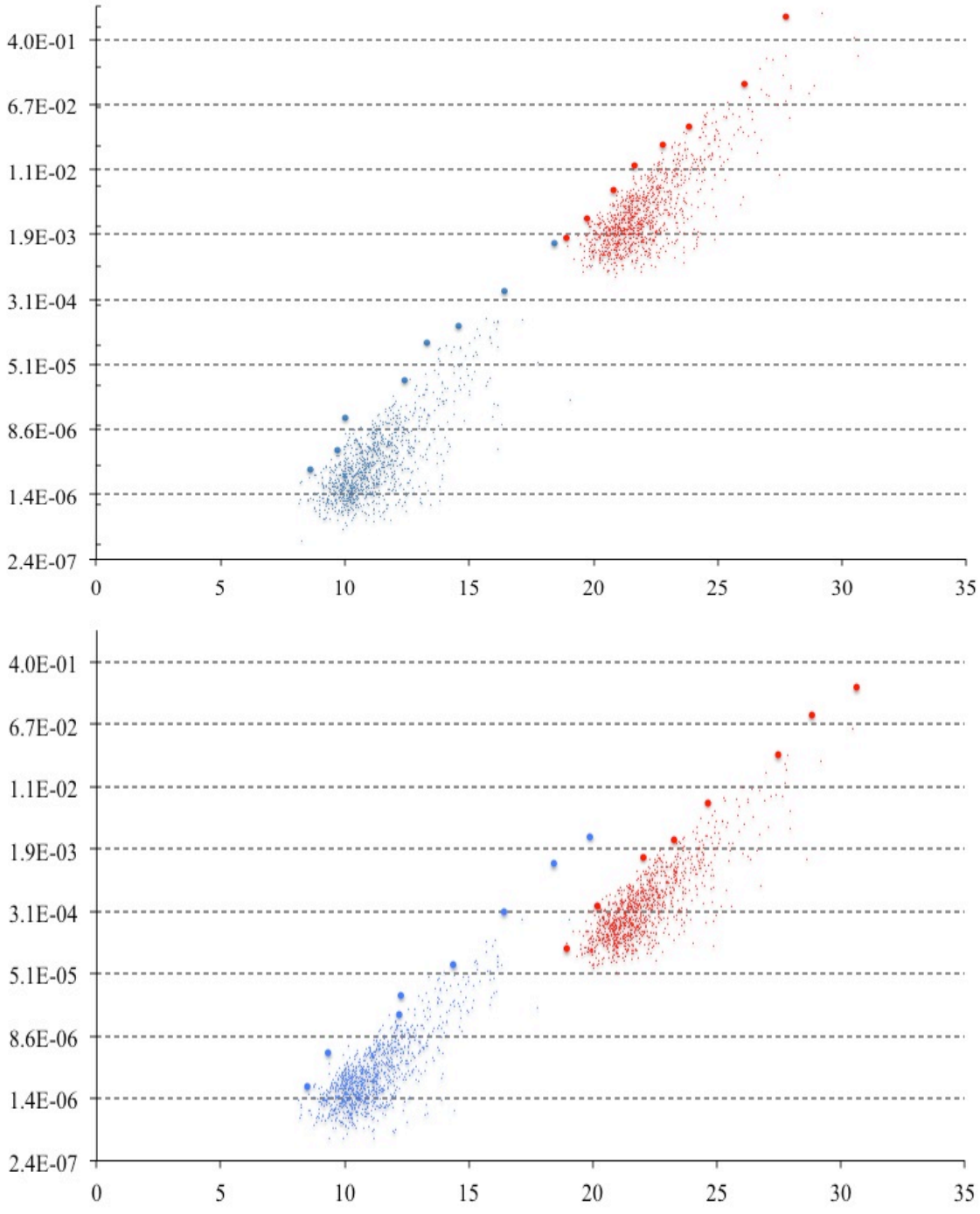

Figure 10. Dependecy on matrix size

To seek a dependency of convergence rates on matrix sizes for LU and QR, we introduce tuned convergence rates as follows:

$$\Omega_{QR\text{-}Tuned}(\kappa(A)) = n^{0.5} \times \Omega_{QR}(n, \kappa(A)) \text{ and } \Omega_{LU\text{-}Tuned}(\kappa(A)) = \Omega_{LU}(n, \kappa(A)).$$

By this tuning, the convergence rate is finally a function of condition numbers. Figure 11 shows the tuned convergence rates for LU (top in Figure 11) and QR (bottom). In the figure, blue dots represent the convergence coefficients for matrix size = 64, orange dots for 128, green dots for 2K, and red dots for 8K. Worst convergence rates are marked with large dots to seek the practical convergence bounds according to condition numbers. The other 4000 results for n = 256, 512, 1K, 4K are not displayed in Figure 11, since they also follow the same tendency shown in Figure 11. Based on the larger dots in Figure 11, we seek practical convergence rate bounds depending on infinity norm based condition numbers. The larger dots in Figure 11 can be interpreted as practical worst convergence rates within 1000 matrices. In other words, it might have an around 99.9 % credibility in average cases. This small 0.1 % may make the bound for convergence rates according to condition numbers significantly different, since theoretical bound could be much larger because of the 0.1%.

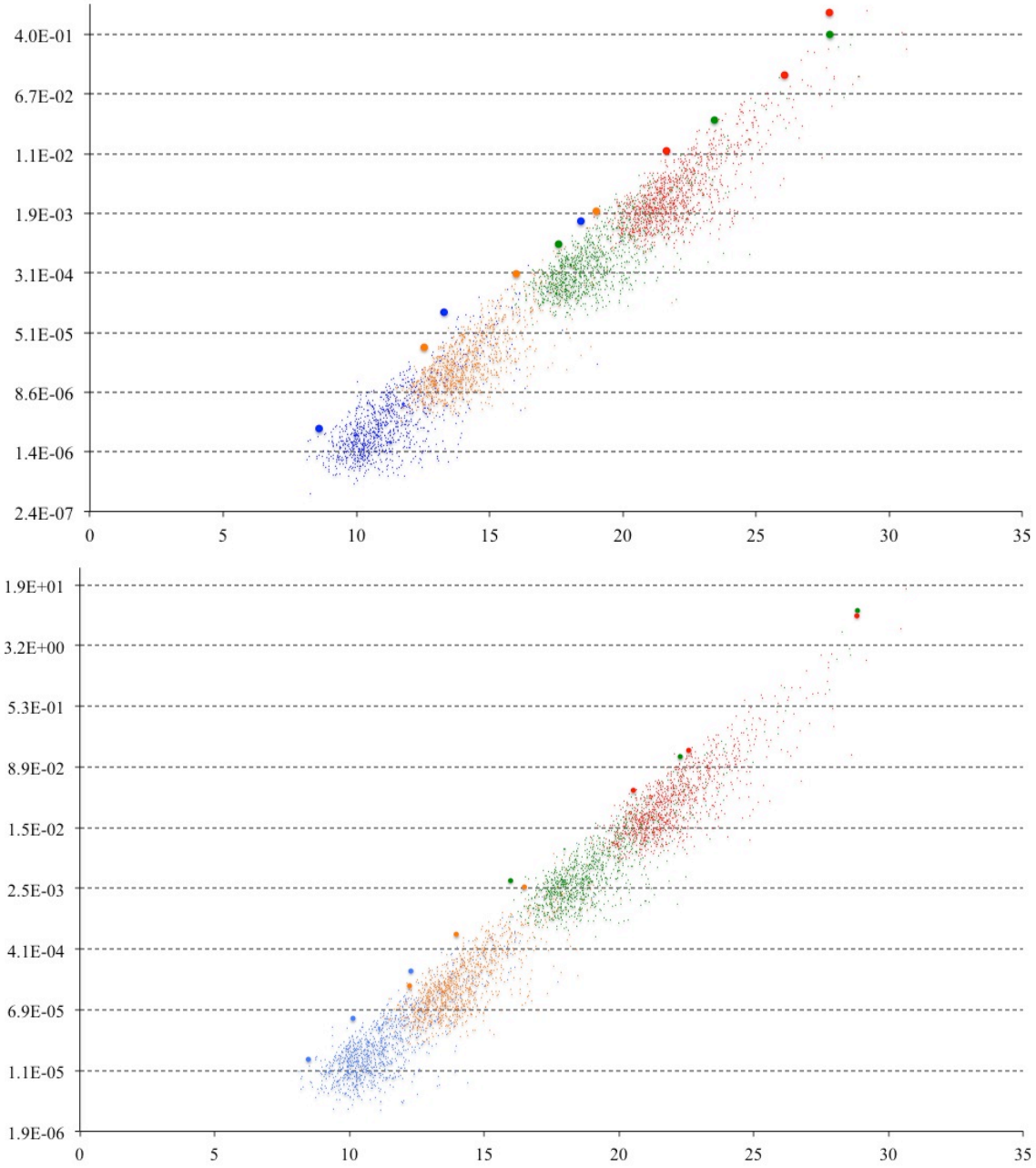

Figure 11. Tuned converge rates for MPIRs

Figure 12 shows practical bounds for convergence rates for MPIRs employing LU and QR. The top figure represents the practical convergence rate bounds for MPIR employing LU, the bottom figure represents the practical convergence rate bounds for MPIR employing QR. In the figure, by regressing method, the relations between the worst convergence rates and the infinity norm based condition numbers are specified with R squared values, which represent the strength of dependability between the two factors (i.e., infinity norm condition numbers and convergence rates). For example, if the R squared value is '1', the infinity norm condition numbers are perfectly related with the convergence rates with the equation. Hence, the relations are very reliable since the R squared value for LU is 0.9956 and 0.9985 for QR. Practical convergence rate bounds depending on matrix sizes and infinity norm based condition numbers can be represented as follows:

$$\Omega_{LU}(n, \kappa(A)) = \Omega_{LU\text{-}Tuned}(\kappa(A)) = 10^{-8}\,\kappa(A)^{0.8989} < 10^{-8}\,\kappa(A)$$

$$\Omega_{QR}(n, \kappa(A)) = \Omega_{QR\text{-}Tuned}(\kappa(A))/\sqrt{n} = 7\times10^{-8}\,\kappa(A)^{0.9308}/\sqrt{n} < 7\times10^{-8}\,\kappa(A)/\sqrt{n}.$$

The bounds for the practical convergence rate are measured using single precision. It would be useful to seek for the practical error bounds depending on lower precisions. We already found empirically that an applied lower precision has a linear relation with convergence rate. Hence, Dividing the practical worst convergence bounds by single precision can estimate the bounds for the practical convergence rates depending on a lower precision according to infinity norm condition numbers and matrix sizes for average cases as Property 4.3 (P 4.3).

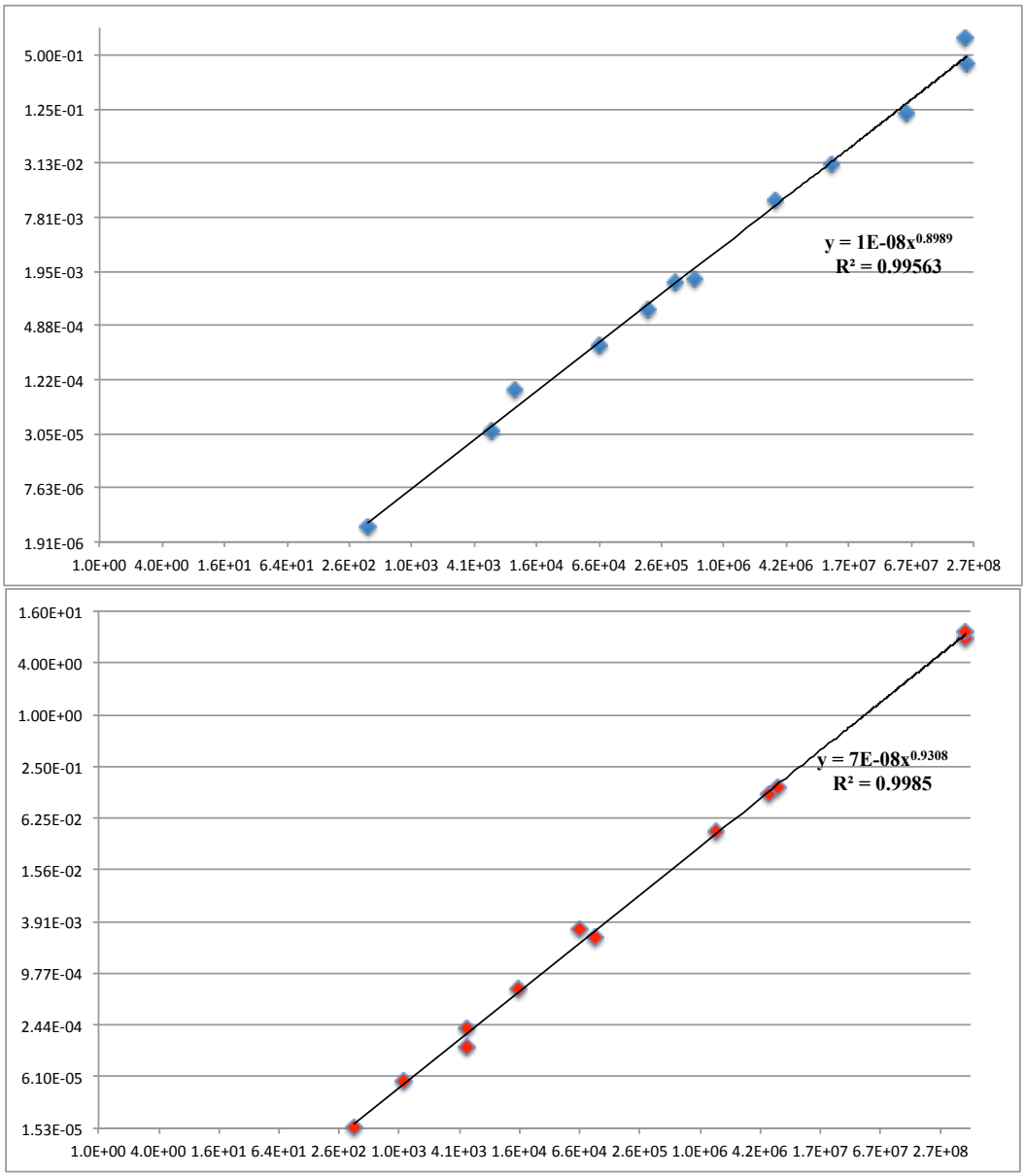

Figure 12. Practical convergence rate bounds for MPIRs (T:LU, B:QR)

Property 4.3 (P 4.3): The empirical convergence rates for Gaussian random matrices depending on matrix sizes and condition numbers are:

$$\Omega_{LU}(n, \kappa(A), \epsilon_L) = \Omega_{LU}(n, \kappa(A))\, \epsilon_L / 2^{-24} < 0.17\, \kappa(A)\, \epsilon_L$$

$$\Omega_{QR}(n, \kappa(A), \epsilon_L) = \Omega_{QR}(n, \kappa(A))\, \epsilon_L / 2^{-24} < (1.18/\sqrt{n})\, \kappa(A)\, \epsilon_L.$$

4-4. Practical choice for a lower precision

To seek appropriate lower precisions for LU and QR, we first explore the tuned worst convergences, $\Omega_{Tuned}(\kappa(A)) = n^k\, \Omega(n, \kappa(A), \epsilon_L) = \phi_{Tuned}(\kappa(A))\, \epsilon_L$, where k = 0 for LU and 0.5 for QR. For successful convergence, the practical convergence bound should be less than 1 as follows : $\Omega(n, \kappa(A), \epsilon_L) = n^{-k}\, \phi_{Tuned}(\kappa(A))\epsilon_L < 1$. Hence, $\epsilon_L < n^k\, \phi_{Tuned}(\kappa(A))^{-1}$. For LU, $\epsilon_{L\_LU}(\kappa_\infty(A)) < (0.17\, \kappa(A))^{-1}$ and for QR, $\epsilon_{L\_QR}(n, \kappa(A)) < \sqrt{n}\, (1.18\, \kappa(A))^{-1}$. Since most computers employ radix 2 based system [70], we try to find out appropriate mantissa bit width for a practical lower precision for MPIR employing LU as follows:

$$-\log_2(\epsilon_{L\_LU}(\kappa(A))) > -\log_2(0.17\, \kappa(A))^{-1}$$

In rounding modes, the mantissa bit width t should follow: $t_L + 1 \geq -\log_2(\epsilon_L)$ to satisfy numeric accuracies for floating point operations. Therefore,

$$t_{L\_LU} + 1 > -2 + \log_2\kappa(A) > \log_2(0.17) + \log_2\kappa(A),$$

$$t_{L\_LU} = \text{minimum integer } t, \text{ satisfying } t > -3 + \log_2\kappa(A).$$

In case of QR,

$$-\log_2(\epsilon_{L\_QR}(n, \kappa(A))) > -\log_2(n^{0.5}\, (1.18\, \kappa(A))^{-1}),$$

$$t_{L\_QR} + 1 > -0.5\log_2 n + 0.3 + \log_2\kappa(A),$$

$$t_{L\_QR} = \text{minimum integer } t, \text{ satisfying } t > -0.5\log_2 n + \log_2\kappa(A).$$

For examples, SDIR employing LU should satisfy: $23 > -3 + log_2 \kappa(A)$ and $\kappa(A) < 2^{26}$, which is a little bit larger than the reciprocal of machine precision (i.e., 2^{-24}). In the case of QR, if the size of matrix is 2^{20}, the allowable condition number is: $23 > -10 + log_2 \kappa(A)$ and $\kappa(A) < 2^{33}$. Hence, QR can be recommended for a matrix possessing a large condition number.

4-5. Summary of chapter

Wilkinson states that the iterative refinement success depends on the backward errors (i.e., ||E||/||A||, see chapter 2). If the infinite precisions are applied for steps 2 and 4, the success condition is as follows: $||A|| \; ||A^{-1}|| < 0.5||A||/||E||$. The success convergence conditions depend on the norm of the matrix ||A||. The backward error E grows with O(n) according to matrix sizes for LU and $O(n^{1/2})$ for QR in average cases [64]. Based on this reasoning, the success for iterative refinement depends on the matrix norm growth according to its sizes for GRMs, since the norms of GRMs also grow according to matrix sizes. Based on our experiments, the relation between infinity matrix norm and matrix size for GRMs is : $n \approx 1.22 \; ||A|| - 33.5$ when $n \geq 64$. Therefore, the effect from the matrix size growth for success condition is approximately diminished for LU – since both ||E|| and ||A|| grow with O(n), ||A||/||E|| is almost O(1). The allowable condition number for QR becomes larger when matrix sizes grow since ||A||/||E|| becomes larger with $O(n^{1/2})$ in GRMs.

We found empirically that the infinity norm based condition numbers measured by LAPACK has a relation with 2 norm based condition numbers as follows: $\kappa_\infty(A) \approx \sqrt{n}$

$\kappa_2(A)$. Hence, if we apply the 2 norm based condition numbers instead of infinity norm based condition numbers, the convergence rates for QR could be independent of matrix sizes while the convergence rates for LU could be dependable on matrix sizes. The dependency on matrix sizes can be explained by growth factors caused by LU. In GRMs, the growth factor is approximately $O(n^{1/2})$ for LU and O(1) for QR.

Chapter 5

Adaptive dynamic precision iterative refinement

When iterative refinement was firstly proposed, it was recommended to employ a higher precision in step 2 (e.g., HPIR) since the residual contains critical information for convergence [36, 37]. Recently, some literatures [28, 31, 34] have shown that OPIR is also able to converge to the exact solution. Now, here we discuss a more aggressive method for an iterative refinement with less run times than the OPIR. We name the method AIR, which is acronym for Adaptive Dynamic Precision Iterative Refinement. This chapter introduces AIR algorithm and discusses numeric correctness and run time of AIR.

5-1. Algorithm for adaptive dynamic precision iterative refinement

The idea of iterative refinement is to reduce the error in the solution gradually per iteration in proportion to the convergence rate. The error contains the relative error in a computed solution from a direct method (i.e., relative error in step 3) and round-off errors in steps 2 and 4. If the rounding errors in steps 2 and 4 are relatively smaller than the relative errors in step 3, the convergence rate is mainly determined in solving triangular systems. In this case, it is not necessary to provide an extremely high precision in step 2, since the error after the truncation may be significantly large compared to rounding errors caused in step 2. Hence, it would be worthy to explore an appropriate precision for steps 2 and 4 at corresponding iteration.

An iterative refinement on FPGAs can determine the appropriate precisions for steps 2 and 4 *adaptively* according to the convergence rates. To determine the least sufficient precisions for steps 2 and 4, we explore the resultant residual internally in terms of register level. Figure 13 describes how to determined the least sufficient precisons adaptively in iterative refinement. In Figure 13, if $\| r^{(i)}\|_\infty = 1$ and $\| r^{(i+1)}\|_\infty = 2^{-5}$, the value for the exponent for $\| r^{(i)}\|_\infty$ is '0' excluding bias (i.e., in this case, the actual value in the register is the bias for IEEE 754) and the exponent for $\|r_j^{(i+1)}\|_\infty$ should be '-5'. If we employ the precision for step 2 as many as the number of cancellation bits, there does not exist truncation error, since the truncated bits are represented as all 0s. Therefore, the least sufficient precision requires 5 bits more compared to the precision for the previous iteration in Figure 13.

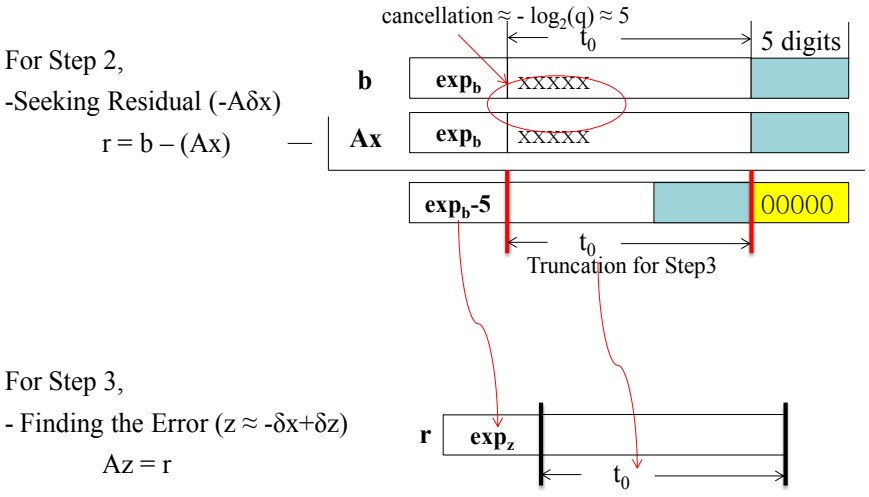

Figure 13. Precisions for step 2 and 4

We can *adaptively* choose an appropriate precision based on the number of cancellation bits, but in actuality, the number of cancellation bits on each component of a residual vector is different. Hence, we determine an appropriate precision based on infinity norm-wise residuals instead of component-wise residuals for AIR. This determination makes possible for users to decide the bit-stream file corresponding to a precision for next iteration. This norm wise determination is examined empirically in section 5-5. Convergence rates according to iterations are not a constant but variable depending on condition numbers, right side vector direction and extra factors. However, the variance is not significant in practice so that we may assume that the current convergence rate is similar with the previous one.

AIR increases the number of mantissa bits for steps 2 and 4 as much as required number of bits according to convergence rates. If we attach the mantissa bits based on recent convergence rates, the precision for steps 2 and 4 is estimated as: $\epsilon^{(i)} = \bar{\sigma}^{(i)} \epsilon^{(i-1)}$, where $\bar{\sigma}^{(i)} = \sigma^{(i-1)}\sigma^{(i-1)}\sigma^{(i-2)}..\sigma^{(1)}$ and $\sigma^{(i)}$ is the convergence rate at the i^{th} iteration based on infinity norm wise residuals. Determining the precision $\epsilon^{(i)}$ requires the assumption that the previous convergence rate is same as current convergence rate (i.e., two $\sigma^{(i-1)}$s). We also assume a lower precision is appropriately chosen to make AIR work (i.e. the relative norm wise error for step 3 is less than unity). The norm-wise determiniation for precision for AIR is described in Algorithm III. Currently, AIR is designed to produce a prescribed accuracy in backward error. Notice that doubled initial precision is applied only for the first iteration since we do not know the convergence rate before experiencing refinement procedure.

Algorithm III. Adaptive dynamic precision iterative refinement :

The adaptively chosen mantissa widths for steps 2 and 4 :

$t^{(1)} = 2\ t_L$ if $t_A > 2\ t_L$, or $t^{(1)} = t_A$ if $t_A \leq 2\ t_0$,

$t^{(2)} = t_L + 2 \times \lceil \log_2 (\|b\|_\infty / \|r^{(1)}\|_\infty) \rceil$,

$t^{(3)} = t_L + \lceil \log_2 (\|b\|_\infty / \|r^{(2)}\|_\infty) \rceil + \lceil \log_2 (\|r^{(1)}\|_\infty / \|r^{(2)}\|_\infty) \rceil$,

$t^{(i)} = t_L + \lceil \log_2 (\|b\|_\infty / \|r^{(i-1)}\|_\infty) \rceil + \lceil \log_2 (\|r^{(i-2)}\|_\infty / \|r^{(i-1)}\|_\infty) \rceil$,

if $t^{(i)} > t_A$, $t^{(i)} = t_A$.

Note.

$t^{(i)}$: the required mantissa bits at the i^{th} iteration
t_L : the required mantissa bits for the lower precision
t_A : the required mantissa bits for the original precision

The numeric accuracy for Algorithm III is tested in section 5-5. Since log_2 operations require considerable run time, a practical algorithm for Algorithm III is discussed in chapter 6.

5-2. Correctness of adaptive dynamic precision iterative refinement

The proof for correctness of AIR adapts the proof used in [31]. We perform numeric analysis of AIR assuming there exist no round-off errors on step 4 at first to understand the idea of AIR with ease. We perform a numeric analysis considering the round-off errors in step 4 after that. We assume that the increased number of bits is equal to the number of cancellation bits in AIR. Consequently, the truncation error from step 2 to 3 is not considered. In step 2, there is an arbitrary error δy from the matrix vector multiplication.

$$r^{(i)} = (I + \delta I^{(i)})\,(b - Ax^{(i)} + \delta y^{(i)}) = (I + \delta I^{(i)})\,(-A\delta x^{(i)} + \delta y^{(i)}),$$

$$\| r^{(i)} \| \le (1 + \epsilon^{(i)})\,(\|A\|\,\|\delta x^{(i)}\| + \|\,\delta y^{(i)}\|), \text{ where } \|\delta y^{(i)}\| \le c_1\,\|A\|\,\|x^{(i)}\|\;\epsilon^{(i)}.$$

In step 3,

$$(A + \Delta A)\,(z^{*(i)} + \delta z^{(i)}) = r^{(i)}$$

where $z^{(i)*} = A^{-1}r^{(i)} = (-\delta x^{(i)} + A^{-1}\delta y^{(i)}) + A^{-1}\,\delta I^{(i)}\,(-A\delta x^{(i)} + \delta y^{(i)})$, $\|\delta z^{(i)}\| \le q\,\|z^{*(i)}\|$, and $q < 1$. In step 4, we assume that there exists no round-off error (we will consider the round-off error in step 4 shortly).

$$x^{(i+1)} = x^{(i)} + z^{(i)} = x^{*} + \delta x^{(i)} + z^{(i)*} + \delta z^{(i)} \tag{5.1}$$

$$= x^{*} + A^{-1}\delta y^{(i)} + A^{-1}\delta I^{(i)}(-A\delta x^{(i)} + \delta y^{(i)}) + \delta z^{(i)} = x^{*} + \delta x^{(i+1)}$$

where $\delta x^{(i+1)} = A^{-1}\delta y^{(i)} + A^{-1}\,\delta I^{(i)}\,(-A\delta x^{(i)} + \delta y^{(i)}) + \delta z^{(i)}$.

Therefore,

$$r^{(i+1)} = (I + \delta I^{(i+1)})\,(b - Ax^{(i+1)} + \delta y^{(i+1)})$$

$$= (I + \delta I^{(i+1)})\,(\delta y^{(i+1)} - \delta y^{(i)} - \delta I^{(i)}\,(-A\delta x^{(i)} + \delta y^{(i)}) - A\delta z^{(i)}),$$

$$\|r^{(i+1)}\| \le (1 + \epsilon^{(i+1)})\|A\|\,\|\delta z^{(i)}\| + \|\delta y^{(i+1)} - \delta y^{(i)} + \delta I^{(i)}A\delta x^{(i)}\| + O(\epsilon^2)$$

$$\le q\,(1 + \epsilon^{(i+1)})\|A\|\,\|z^{*(i)}\| + \|\delta y^{(i+1)} - \delta y^{(i)} + \delta I^{(i)}A\delta x^{(i)}\| + O(\epsilon^2)$$

$$\le q\,(1 + \epsilon^{(i+1)})\|A\|\,(\|\delta x^{(i)}\| + \|A^{-1}\|\,\|\delta y^{(i)}\| + \epsilon^{(i)}\,(\|A^{-1}\|\,\|A\|\,\|\delta x^{(i)}\| + \|\delta y^{(i)}\|)) + \|\delta y^{(i+1)} - \delta y^{(i)}$$

$$+ \delta I^{(i)}A\delta x^{(i)}\| + O(\epsilon^2)$$

$$\le q\,\|A\|\,(\|\delta x^{(i)}\| + \|A^{-1}\|\,\|\delta y^{(i)}\| + \epsilon^{(i)}\,\|A^{-1}\|\,\|A\|\,\|\delta x^{(i)}\|) + q\,\epsilon^{(i+1)}\,\|A\|\,\|\delta x^{(i)}\| + \|\delta y^{(i+1)} - \delta y^{(i)}$$

$$+ \delta I^{(i)}A\delta x^{(i)}\| + O(\epsilon^2)$$

$$\le\; (q + (\kappa(A) + 1)\epsilon^{(i)} + q\,\epsilon^{(i+1)})\,\|A\|\,\|\delta x^{(i)}\| + (q\,\kappa(A) + 1)\,\|\delta y^{(i)}\| + \|\delta y^{(i+1)}\| + O(\epsilon^2). \tag{5.2}$$

Considering round-off errors in step 4, we derive the forward error in AIR described in Theorem 5.1.

Theorem 5.1 (Th 5.1): The forward error at i^{th} iteration in AIR is represented as :

$$|| \delta x^{(i+1)} ||/|| x^* || \leq (\prod_{k=1}^{i} \sigma^{(k)})\, q + p^{(i)} + \sum_{j=i}^{2}(\prod_{l=i}^{j} \sigma^{(l)})\, p^{(j-1)} + O(\epsilon^2)$$

where $\sigma^{(i)} = q + (1 + (1+q)\ (1+ \kappa(A)(1+c_i))\ )\epsilon^{(i)}$, $p^{(i)} = (c_i\, \kappa(A)\ (1+q) + 1)\epsilon^{(i)}$, c_i is a constant depending on matrix size and the corresponding iteration.

Proof: From (5.1), we need to put some quantity representing round-off errors $\delta d^{(i)}$ in step 4 as : $x^{(i+1)} = x^{(i)} + z^{(i)} + \delta d^{(i)} = x^* + \delta x^{(i)} + z^{(i)*} + \delta z^{(i)} + \delta d^{(i)}$

$= x^* + A^{-1}\delta y^{(i)} + A^{-1}\ \delta I^{(i)}\ (-A\delta x^{(i)} + \delta y^{(i)}) + \delta z^{(i)} + \delta d^{(i)} = x^* + \delta x^{(i+1)}$

where $\delta x^{(i+1)} = A^{-1}\delta y^{(i)} + A^{-1}\ \delta I^{(i)}(-A\delta x^{(i)} + \delta y^{(i)}) + \delta z^{(i)} + \delta d^{(i)}$ and $||\delta d^{(i)}|| \leq || x^* + \delta x^{(i)} + z^{(i)*} + \delta z^{(i)} ||\ \epsilon_i$. Therefore,

$$|| \delta x^{(i+1)} || \leq ||A^{-1}||\ ||\delta y^{(i)}|| + \epsilon^{(i)}\ ||A^{-1}||\ ||\ -A\delta x^{(i)} + \delta y^{(i)}\ ||\ + q\ ||z^{*(i)}|| + ||\delta d^{(i)}||$$

$\leq \epsilon^{(i)}\ \kappa(A)\ \ ||\delta x^{(i)}|| + (1+\epsilon^{(i)})\ ||A^{-1}||\ ||\delta y^{(i)}\ || + (q(1+\ \epsilon^{(i)}) + \epsilon^{(i)})\ ||z^{*(i)}|| + \epsilon^{(i)}\ ||\ x^* + \delta x^{(i)}\ ||$

$\leq ((q(1+\ \epsilon^{(i)}) + \epsilon^{(i)})\ (1+\ \epsilon^{(i)}\kappa(A)\ ) + \epsilon^{(i)}\ \kappa(A)\ \ )\ ||\delta x^{(i)}|| + [(1+\ \epsilon^{(i)})(\ 1+(q(1+\ \epsilon^{(i)}) + \epsilon^{(i)})\ )\ c_1\ \kappa(A)\ \epsilon^{(i)} + \epsilon^{(i)}\ ]\ ||\ x^* + \delta x^{(i)}||$

$= (\ q + \epsilon^{(i)}(1+q)(1+\ \kappa(A)\ )\ )\ ||\delta x^{(i)}|| + (c_1\ \kappa(A)\ (1+q) + 1)\epsilon^{(i)}\ ||\ x^* + \delta x^{(i)}|| + O(\epsilon^2)$,

$= (q + (1+(1+q)(1+\ \kappa(A)\ (1+c_1)))\epsilon^{(i)})\ ||\delta x^{(i)}|| + (c_1\ \kappa(A)\ (1+q) + 1)\epsilon^{(i)}\ ||\ x^*|| + O(\epsilon^2)$,

since $||z^{*(i)}|| = ((1+\ \epsilon^{(i)}\kappa(A)\ )\ ||\delta x^{(i)}|| + (1+\ \epsilon^{(i)})\ ||A^{-1}\ ||\ ||\delta y^{(i)}||)$. Hence,

$|| \delta x^{(i+1)} ||/|| x^* || \leq (q + (1+(1+q)(1+\ \kappa(A)\ (1+c_1)))\epsilon^{(i)})\ ||\delta x^{(i)}||/|| x^* || + (c_1\ \kappa(A)\ (1+q) + 1)\epsilon^{(i)} + O(\epsilon^2)$.

Let $\sigma^{(i)} = q + (1+(1+q)(1+ \kappa(A)\ (1+c_i)))\epsilon^{(i)}$ and $p^{(i)} = (c_i\, \kappa(A)\ (1+q) + 1)\epsilon^{(i)}$,

$|| \delta x^{(i+1)} ||/|| x^* || \leq \sigma^{(i)}\ ||\delta x^{(i)}||/|| x^* || + p^{(i)} + O(\epsilon^2)$.

Using recursive relation,

$|| \delta x^{(i+1)} ||/|| x^* || \leq \sigma^{(i)}\ (\sigma^{(i-1)}\ (\sigma^{(i-2)}\ ||\delta x^{(i-2)}||/|| x^* || + p^{(i-2)}) + p^{(i-1)}\ ) + p^{(i)} + O(\epsilon^2)$

$= (\prod_{k=1}^{i} \sigma^{(k)})\ ||\delta x^{(1)}||/||x^*|| + p^{(i)} + \sigma^{(i)}\ p^{(i-1)} + \sigma^{(i)}\ \sigma^{(i-1)}\ p^{(i-2)} + \ \ldots + (\prod_{k=2}^{i} \sigma^{(k)}\ )p^{(1)} + O(\epsilon^2)$

$= (\prod_{k=1}^{i} \sigma^{(k)})\ q + p^{(i)} + \sum_{j=i}^{2}(\prod_{l=i}^{j} \sigma^{(l)})\ p^{(j-1)} + O(\epsilon^2)$. Q. E. D.

Difference between MPIR and AIR in terms of accuracy and convergence rate comes from the precisions applied. The convergence and accuracy for AIR in terms of infinity norm (i.e. c_3 corresponds to unity in [31]) are equal with Theorem 3.1 in [31], if you arrange the convergence rate $\sigma^{(i)}$ and accuracy $p^{(i)}$ with first order of precision. The mathematically driven convergence rate from FPIR (i.e., σ_1 in [31]) is not function of the number of iteration, but the convergence rate in AIR is a function of the number of iteration. The accuracy σ_2 in [31] is not a function of iteration, but it is in AIR.

5-3. Numerical stability and well-behaved

In this section, we define numerically stable and well-behaved as in [31] and prove AIR also numerically stable and well-behaved if the system is not too ill-conditioned.

Definition 5.1 (Def 5.1) [31]: We also define that AIR is numerically stable if

$$|| x_c - x^* ||/|| x^* || \leq \mu_1\, \kappa(A)\ \ \epsilon_A$$

and is well-behaved if

$$|| b - Ax_c || \leq \mu_2 \epsilon_A ||A|| \, ||x^*||$$

where each μ_1 and μ_2 is a constant depending on the size n respectively.

Theorem 5.2 (Th 5.2): AIR is numerically stable if the system is not too ill conditioned.

Proof: In AIR, we assume that there is no truncation error from step 2 to 3 (refer to section 5-1). In Theorem 5.1,

$$|| \delta x^{(i+1)} ||/|| x^* || \leq (\prod_{k=1}^{i} \sigma^{(k)}) \, q + p^{(i)} + \sum_{j=i}^{2} (\prod_{l=i}^{j} \sigma^{(l)}) \, p^{(j-1)} + O(\epsilon^2).$$

If $\max_w(\sigma^{(w)})$ is less than unity, the term $(\prod_{k=1}^{i} \sigma^{(k)}) \, q$ approaches to zero when we increase the number of iterations. The precision for step 2 can reach the original precision, if $\sigma^{(i)} = (q + (\kappa(A) + 1)\epsilon^{(i)} + q \, \epsilon^{(i+1)}) \leq ½$, which means the condition number times the precision applied for step 2 is relatively smaller than unity (i.e, the system is not too ill-conditioned). Assuming that the system is not too ill conditioned,

$p^{(i)} = (c_i \, \kappa(A) \, (1+q) + 1)\epsilon_A$. If we put $\bar{c} = \max_j (c_j \, \kappa(A) \, (1+q) + 1)$,

$\sum_{j=i}^{2} (\prod_{l=i}^{j} \sigma^{(l)}) \, p^{(j-1)} \leq \bar{c} \sum_{j=i}^{2} (\prod_{l=i}^{j} \sigma^{(l)}) \;\; \epsilon^{(j-1)} = \bar{c}(\sigma^{(i)}\epsilon^{(i-1)} + \sigma^{(i)}\sigma^{(i-1)}\epsilon^{(i-2)} + \ldots + (\prod_{j=2}^{i} \sigma^{(j)}) \, \epsilon^{(1)})$.

When the precision for steps 2 and 4 in AIR reaches ϵ_A, there can be possible further cancellation error to reduce the previous round-off errors. Letting $\max_w(\sigma^{(w)}) = \tilde{\sigma} \leq ½$ (i.e., there is a cancellation error in step 2),

$\lim_{i \to \infty} \sum_{j=i}^{2} (\prod_{l=i}^{j} \sigma^{(l)}) \, p^{(j-1)} \leq \;\; \bar{c} \, \tilde{\sigma} \, \epsilon_A / \, (1 - \tilde{\sigma}) + \lim_{i \to \infty} \tilde{\sigma}^{\,i-k} \epsilon^{(1)} \leq \;\; \bar{c} \, \epsilon_A$, where k is the number of iteration before the applied precision is less than ϵ_A. Hence,

$|| \delta x^{(i+1)} ||/|| x^* || \le 2\bar{c}\, \epsilon_A + O(\epsilon^2) = 2(\max_j(c_j)(1+q) + 1/\kappa(A)\,)\, \kappa(A)\ \epsilon_A + O(\epsilon^2)$. Q. E. D.

Based on (Def 5.1), AIR is numerically stable. The achievable accuracy is at most original precision accuracy magnified by the condition number.

Now, we examine the well behaved defined in (Def 5.1) for AIR. Well behaved implies numerically stable, but the other way around [31]. AIR is well behaved as described in Theorem 5.3.

Theorem 5.3 (Th 5.3): AIR is well behaved if the system is not too ill-conditioned (i.e., if $q\ \kappa(A)$ is at most order of unity and $max_i(\sigma^{(i)})$ is less than one half).

Proof: Based on (5.2), at the last iteration,

$$|| b - Ax^{(i+1)} || = ||r^{(i+1)} - \delta y^{(i+1)} + \epsilon_A\,(b - Ax^{(i+1)} + \delta y^{(i+1)}) ||$$

$$\le (q + (\kappa(A) + 1)\epsilon^{(i)} + q\,\epsilon^{(i+1)})\, ||A||\, ||\delta x^{(i)}|| + (q\,\kappa(A) + 1)\, ||\delta y^{(i)}|| + \epsilon_A\, ||b - Ax^{(i+1)}|| + O(\epsilon^2),$$

$$\le (q + (\kappa(A) + 1)\epsilon^{(i)} + q\,\epsilon^{(i+1)})\, ((c_{i-1}\,(1+q) + (1+\bar{c}\,)/\,\kappa(A)\,)\,\kappa(A)\,\epsilon_A)\, ||A||\, ||x^*|| +$$

$$(q\,\kappa(A) + 1)\, c_i\, ||A||\, ||x^{(i)}||\, \epsilon^{(i)} + \epsilon_A\, ||b - Ax^{(i+1)}|| + O(\epsilon^2),$$

$$\le (q + (\kappa(A) + 1)\epsilon^{(i)} + q\,\epsilon_A)\, (c_{i-1}\,(1+q)\,\kappa(A)\ \epsilon_A + (1+\bar{c}\,)\epsilon_A)\, ||A||\, ||x^*|| +$$

$$(q\,\kappa(A) + 1)\, c_i\, ||A||\, ||x^{(i)}||\, \epsilon^{(i)} + \epsilon_A\, ||b - Ax^{(i+1)}|| + O(\epsilon^2),$$

$$|| b - Ax^{(i+1)} || \le \{ (q + (\kappa(A) + 1)\epsilon^{(i)} + q\,\epsilon_A)\, (c_{i-1}\,(1+q)\,\kappa(A)\ \epsilon_A + (1+\bar{c}\,)\epsilon_A)\, ||A||\, ||x^*|| + (q\,\kappa(A) + 1)\, c_i\, ||A||\, ||x^{(i)}||\, \epsilon^{(i)} \}\, /(1 - \epsilon_A) + O(\epsilon^2),$$

Since $\lim_{i\to\infty} \epsilon^{(i)} = \epsilon_A$, $|| x^{(i)}|| \le || x^*|| + || \delta x^{(i)}||$, and $1 >> \epsilon_A$,

$$\le (q\,(\bar{c}\,(1+q)\,\kappa(A) + 1 + c') + \bar{c}\,q\,\kappa(A) + \bar{c}\,)\,\epsilon_A\, ||A||\, ||x^*|| + O(\epsilon^2)$$

$= (\bar{c} + \bar{c}\, q\, \kappa(A)\ (2 + q) + q(1 + \bar{c}\,))\, \epsilon_A\, ||A||\, ||x^*|| + O(\epsilon^2)$. Q. E. D.

Based on Theorem 5.2 and 5.3, AIR is numerically stable and well-behaved though we employ lower precisions than original precision for steps 2 and 4.

The required number of iterations in AIR could be larger than OPIR, since AIR employs fewer mantissa widths than OPIR for steps 2 and 4. The required number of iterations for AIR is described in Theorem 5.4.

Theorem 5.4 (Th 5.4): The number of iterations in AIR requires at most one more iteration than OPIR if the system is not too ill conditioned.

Proof: In Theorem 5.1, $||\delta x^{(i+1)}||/||x^*|| \leq (\prod_{k=1}^{i} \sigma^{(k)})\, q + p^{(i)} + \sum_{j=i}^{2} (\prod_{l=i}^{j} \sigma^{(l)})\, p^{(j-1)} + O(\epsilon^2)$. Letting $\max_w(\sigma^{(w)}) = \tilde{\sigma}$ and $\sigma^{(i)}$ = *convergence rates in the residuals at* i^{th} *iteration*, if the system is not too ill-conditioned, $\tilde{\sigma} \approx q$. In AIR for a not too ill conditioned, system, $\epsilon^{(i)} = \tilde{\sigma}^i \epsilon^{(i-1)}$. Hence, $||\delta x^{(i+1)}||/||x^*|| \leq \tilde{\sigma}^i q + i \cdot p^{(i)} + O(\epsilon^2)$, where *i* denotes the processed number of iterations. The accuracy (i.e., $m \cdot p^{(i)}$) is approximately function of the number of iteration in AIR. If the number of iteration is small (i.e., convergence rate is good and the gap between the lower precision and original precision is not too far), AIR produces the same level of accuracy with OPIR. If the number of iteration is large in OPIR, AIR may require a few more number of iteration than OPIR. However, if $q < 1/m$ and $\tilde{\sigma} \approx q$ (i.e., q is larger than the required number of iteration in not too ill conditioned system), one more iteration is required compared to OPIR, which is very likely situation. Q. E. D.

5-4. Theoretical run time

The run time discussed in (A 3.2) can be represented as follow: $T(\epsilon) = \alpha (- log_2 \epsilon - 1)^{\beta}$, where β is a increasing time-cost ratio according to mantissa bit width increase, α is a scaling factor, and ϵ is a precision. We ignore the run time for step 4 in AIR since it requires only $(n - 1)$ operations, while step 2 and 3 require $2n^2$ operations each. We consider GEPP for step 1. Therefore, the total execution time for AIR can be described as follows:

$$T_{AIR} =(2/3\ n^3 + 2n^2) \times T(\epsilon_L) \ + 2n^2 (m \times T(\epsilon_L) + T(\epsilon_L^2) + \sum_{j=2}^{m} T(\Omega^j \epsilon_L))$$

$$= (2/3\ n^3 + 2n^2)\ \alpha\ t_L^{\beta} + 2\ n^2\ \alpha \times [\ m \times t_L^{\beta} + 2^{\beta}\ t_L^{\beta} + (-2log_2\ \Omega + t_L)^{\beta} + (-\ 3log_2\ \Omega + t_L)^{\beta} + .. + (-\ m\ log_2\ \Omega + t_L\ )^{\beta}]$$

where ϵ_L is a lower precision applied for matrix decomposition, m is the required number of iterations to obtain the prescribed solution accuracy, and $-log_2\Omega$ represents the number of cancelation bits in step 2. The β may depend on implementation methodology for adders (e.g., ripple-carry, carry-skip, and carry-save adders [67]). If $\beta = 1$ for adders, $\beta \approx 2$ for a multiplier employing the adders. For the practical analysis, we assume that $\beta \in [1, 2]$, since $\beta \approx 1$ for adders in general. In the case of $\beta = 1$,

$$T_{AIR} = 2\ n^2 \alpha\ ((2m + 2 + n/3)\ t_0 - ((m+1)/2 - 1)\ log_2\ \Omega) \text{ if } \beta = 1.$$

To compare it to XMIR, the theoretical performance of XMIR is as follows:

$$T_{XMIR} = 2\ n^2 \alpha\ ((2m + 1 + n/3)\ t_0 - m^2\ log_2\ \Omega) \text{ if } \beta = 1.$$

Notice that $log_2\ \Omega$ is a negative value since $\Omega < 1$. For $\beta = 2$, if we employ the property of $\sum_{j=1}^{m} j^2 = m(m+1)(2m+1) / 6$,

$T_{AIR} = 2\, n^2\alpha\, [\, (2m + 4 + n/3)\, t_L^2 - (\, (m^2+m+2)\, \log_2 \Omega\,)\, t_L + (\, m(m+1)(2m+1)/6 - 1) \times \log_2^2 \Omega]$ *if* $\beta = 2$.

Likewise, to compare to XMIR,

$T_{XMIR} = 2\, n^2\alpha\, [\, (2m + 1 + n/3)\, t_L^2 - (2m^2 \log_2 \Omega)\, t_L + m^3 \log_2^2 \Omega]$ if $\beta = 2$.

The impact of AIR on speed is more significant when β is larger. It may be worthy to look at the impact of AIR on run time for plausible βs (i.e., $\beta = [1, 2]$) in terms of the ratio between initial precision and final precision. To observe the impact according to the ratio, we first seek the run time to produce a satisfactory solution in the forward error for XMIR as follows:

$$T_{XMIR} = 2\, n^3/3 \times T(\epsilon_L) + 2\, n^2 T(\epsilon_L) + 2\, n^2 m \times (T(\epsilon_L) + T(\epsilon_A))$$

$$= 2n^2 T(\epsilon_A)\, ((n/3 + 1 + m)\, (t_L/t_A)^\beta + m\,) = 2n^2 T(\epsilon_A)\, ((n/3 + 1 + m)\, \gamma^\beta + m\,) \qquad (5.3)$$

where γ is the ratio of mantissa bit width between the initial precision and refinement precision in XMIR. Based on (5.3), the total execution time mainly depends on γ and β. The impact of precision of XMIR on speed can be represented according to γ. If γ approaches unity (i.e., $\epsilon_L \approx \epsilon_A$), it does not have much benefit in terms of run time, since it still needs $O(n^3)$ of higher precision operations as shown in (5.4). The benefit could be maximized when γ is approaches to zero (i.e., $\epsilon_L >> \epsilon_A$) as shown in (5.5).

$$T_{XMIR\ \gamma\beta \to 1} = 2n^2\, T(\epsilon_A)\, (n/3 + 2m + 1), \qquad (5.4)$$

$$T_{XMIR\ \gamma\beta \to 0} = 2\, n^2 m\, T(\epsilon_A). \qquad (5.5)$$

If $\gamma \to 0$, XMIR works as if it requires $O(n^2)$ higher precision operations. This case happens when the condition numbers are relatively small and a user requires an

extremely high precision accuracy. Now, we look at the run time of AIR according to γ in β = [1, 2]. To seek the run time for AIR, we need to consider two cases such as $t_A > 2t_L$ for $\gamma \rightarrow 0$ and $t_A \leq 2t_L$ for $\gamma \rightarrow 1$. In the case of $t_A > 2t_L$ and $m \geq 2$ when $\beta = 1$,

$$T_{AIR} = (2n^3/3 + 2n^2) \times T(\epsilon_L) + 2n^2 (m \times T(\epsilon_L) + T(\epsilon_L^2) + \sum_{j=2}^{m} T(\epsilon^{(j)}))$$

$$= 2n^2 T(\epsilon_A) [(n/3 + m + 3) \times T(\epsilon_L) / T(\epsilon_A) + T(\epsilon_A)^{-1} \sum_{j=2}^{m} T(\Omega^{(j)} \epsilon_L)]$$

$$= 2n^2 T(\epsilon_A) [(n/3 + 2m + 2) \times \gamma + (m(m+1)/2 - 1) T(\Omega)/T(\epsilon_A)].$$

Since $T(\Omega)/T(\epsilon_A) = (1 - \gamma)/m$ when $\beta = 1$ (i.e., $T(\epsilon_A) = T(\Omega^m \epsilon_L)$ in AIR),

$$= 2n^2 T(\epsilon_A) ((n/3 + 2m + 2) \times \gamma + ((m+1)/2 - 1/m) (1 - \gamma))$$

$$= 2n^2 T(\epsilon_A) ((n/3 + 3m/2 + 3/2 + 1/m) \times \gamma + (m+1)/2 - 1/m).$$

Therefore, $T_{AIR\ \gamma \rightarrow 0} = n^2 (m + 1 - 2/m^{-1}) T(\epsilon_A)$. (5.6)

Based on (5.5) and (5.6), the speedup by AIR compared to XMIR is as follows:

$$\text{Speedup}_{\text{MAX-AIR}} = \text{MAX}(T_{XMIR\ \gamma \rightarrow 0} / T_{AIR\ \gamma \rightarrow 0}) = 2 / \text{MIN}(1 + m^{-1} - 2/m^{-2}) \leq 2$$

where $m \geq 2$. Therefore, the speedup of AIR is bounded to 2X compared to XMIR regardless of matrix sizes and condition numbers when $\beta = 1$. In the case of $t_A \leq 2t_L$ and $m \geq 1$,

$$TC_{AIR} = (2/3\ n^3 + 2n^2) \times T(\epsilon_L) + 2n^2 (m \times T(\epsilon_L) + T(\epsilon_f) + \sum_{j=2}^{m} T(\epsilon^{(j)}))$$

$$= 2n^2 T(\epsilon_A) [(n/3 + 1 + m) \times T(\epsilon_L) / T(\epsilon_A) + 1 + T(\epsilon_A)^{-1} \sum_{j=2}^{m} (T(\epsilon_L) + jT(\Omega))]$$

$$= 2n^2 T(\epsilon_A) [(n/3 + 1 + m) \times T(\epsilon_L)/T(\epsilon_A) + 1 + (m-1) T(\epsilon_L)/T(\epsilon_A) + (m(m+1)/2 - 1)(T(\Omega)/T(\epsilon_A))]$$

$$= 2n^2 T(\epsilon_A) [(n/3 + 1 + m) \times T(\epsilon_L)/T(\epsilon_A) + 1 + (m-1)\ T(\epsilon_L)/T(\epsilon_A) + ((m+1)/2 - 1/m)(1 - \gamma)]$$

$$= 2n^2 T(\epsilon_A) [(n/3 + 2m + 1/m) \times \gamma + (m+3)/2 - 1/m].$$

Therefore, $T_{AIR\ \gamma\to 1} = 2n^2\ T(\epsilon_A)\ (\ n/3 + 3(m+1)/2\ )$, where $m \geq 1$ and

$T_{AIR\ \gamma\to 1} = 2n^2\ T(\epsilon_A)\ (\ n/3 + 1\ )$, where $m = 0$.

When $\gamma \to 1$, both AIR and XMIR can be considered approximately as FPIR, since the initial precision is approximately same as the final precision. Single iteration is usually enough to obtain a numerically stable solution with GEPP in FPIR [34]. Therefore, AIR and XMIR are almost identical method each other and produce an equal run time, when $m = \{0, 1\}$. Therefore, speedup of AIR over XMIR is in the range [1, 2] when $\beta = 1$. When $\beta = 2$, we consider the case that $t_f > 2t_0$ and $m \geq 2$, since AIR is almost identical procedure practically with XMIR when $t_f < 2t_0$.

$$TC_{AIR} = (2n^3/3 + 2n^2)\ T(\epsilon_L)\ + 2n^2\ (\ m\ T(\epsilon_L) + T(\epsilon_L{}^2) + \sum_{j=2}^{m}\ T(\epsilon^{(j)})\ )$$

$$=(2n^3/3 + 2n^2)\ T(\epsilon_L)\ + 2n^2\ (\ m \times T(\epsilon_L) + 4\ T(\epsilon_L) + \sum_{j=2}^{m}\ T(\Omega^{(j)}\ \epsilon_L)\ )$$

$$=2n^2\ (n/3 + m + 5)\ T(\epsilon_L) + \sum_{j=2}^{m}\ T(\Omega^{(j)}\ \epsilon_L)$$

$$=2n^2\ (n/3 + m + 5)\ T(\epsilon_L) + \alpha^2 \sum_{j=2}^{m}\ (\ -j\log_2\Omega - \log_2\epsilon_L)^2$$

$$= 2n^2\ (n/3 + m + 5)\ T(\epsilon_L) + T(\Omega)m(m+1)(2m+1)/6 + (m\ \log_2\Omega + 1)\ (m - 1)\ T(\epsilon_L) - T(\Omega)$$

$$= 2n^2\ (n/3 + 2m + 4 + m(m-1)\log_2\Omega)\ T(\epsilon_L) + (m(m+1)(2m+1)/6 - 1)T(\Omega)$$

$$=2n^2T(\epsilon_A)((n/3+2m+4+m(m-1)\log_2\Omega)\times\gamma^2+m^{-2}(m(m+1)(2m+1)/6-1)(1-\gamma^2(1+2m\ \log_2\Omega)))$$

$$=2n^2T(\epsilon_A)((n/3+5m/3+7/2+m^{-2}+m^{-1}+(m^2/3-2m+1+2m^{-1})\log_2\Omega)\gamma^2+(m+1)(2m+1)/6m-m^{-2})$$

Therefore, $T_{AIR\ \gamma\to 0} = 2n^2\ ((m+1)(2m+1)/(6m) - m^{-2})T(\epsilon_A)$. The speedup over XMIR can be represented as follows:

$$Speedup_{MAX\text{-}AIR} = MAX(T_{XMIR\ \gamma\to 0}/\ T_{AIR\ \gamma\to 0}) = MAX\ (m^3/(m(m+1)(2m+1)/6 - 1)) \leq 3.$$

Speedup of AIR over XMIR is in the range [1, 3] when β= 2. Because of the pattern of summation rules (e.g., $\sum_{j=1}^{m} j=m(m+1)/2$, $\sum_{j=1}^{m} j^2=m(m+1)(2m+1)/6$, and so on), the maximized speedup of AIR over XMIR is $(\beta+1)X$ as described in Theorem 5.5, where βs are integers.

Theorem 5.5 (Th 5.5): Maximized speedup in AIR over XMIR is:

$\text{Speedup}_{\text{MAX}} = (\beta+1)X$, where βs are integers.

In next section, we verify the theoretical correctness and run time for AIR based on our software demo.

5-5. Tests for numeric accuracy and run time

In this section, we test AIR in terms of numeric accuracy and run time. For the tests, we choose GRMs, since GRMs are used for some applications and it is a good example to reflect average cases [75]. Infinity condition numbers are used for the experiments. Four types of arbitrary precision floating point operations such as addition, subtraction, multiplication and division are implemented in C for the tests. Arbitrary precision addition requires three inputs such as two operands and a prescribed mantissa width. The prototype for the addition is described as follows:

double my_add (double a, double b, int mant).

In the prototype, the two double precision operands are initially taken and their mantissa bits are truncated according to a prescribed mantissa width. The two operands are

converted back to double precision floating point formats for floating point arithmetic operations. Once the addition is complete, the mantissa bits of the result are truncated to a prescribed mantissa bit width again. The other three functions are implemented the same way as the addition. Based on the arbitrary precision arithmetic functions, GEPP and triangular matrix solvers are implemented. The implementations are verified by comparing the solutions to the ones from LAPACK (i.e. dgetrs and dgetrf). Implementation for emulation of AIR employs those arbitrary precision functions.

First, the numbers of cancelation bits are explored. Figure 14 represents the numbers of cancelation bits for all the components of a residual vector. In the test, the matrix size is 64 × 64 and 13 bits are employed for the mantissa bit width for steps 1 and 3 and 52 bits for steps 2 and 4. The component wise cancelation bits are mirrored each other based on the center on the average convergence rate (i.e. around 2^{-11}). Based on the figure, choosing a precision based on component-wise cancelation bits is not effective in AIR. For example, if we take 20 more mantissa bits for a precision in 3rd iteration based on 39th component in Figure 14, other components does not obtain benefit form that (i.e., 11 bits are sufficient for most components).

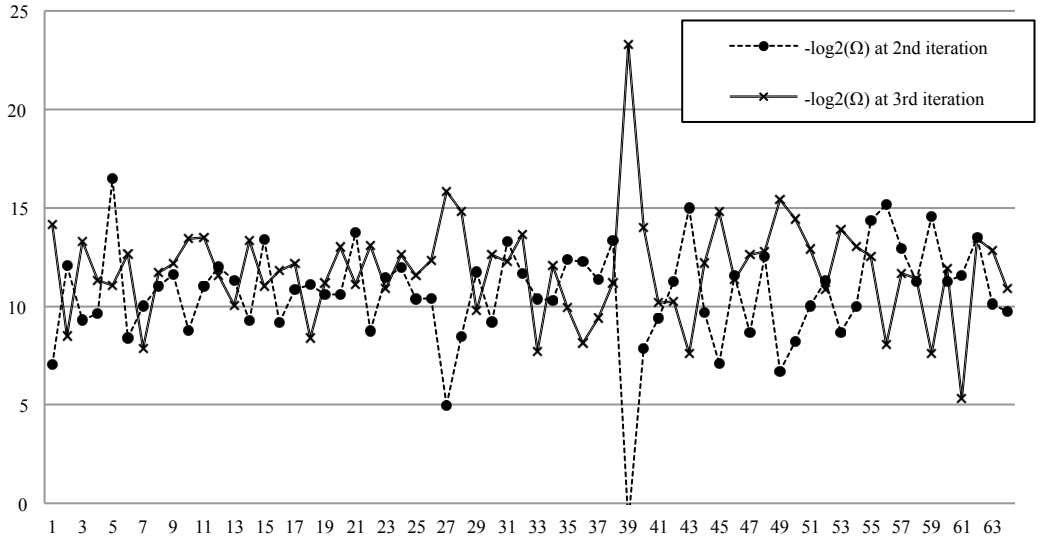

Figure 14. Convergence rates at consecutive two iterations

Figure 15 represents comparison between OPIR and AIR in terms of the number of cancellation bits when AIR considers norm-wise cancelation bits to determine the next mantissa bit width for step 2. Dashed line represents the total number of cancellation bits for AIR when 27, 38, 51 and 52 bits are employed for steps 2 and 4 for 1st, 2nd, 3rd, and 4th iteration. Solid line represents the total number of cancelation bits for OPIR employing 52 bits for step 2 and 4 for whole iteration. No marks on components in Figure 15 specify that the components of the residual are zeros. Based on Figure 15, the two iterative refinements produce almost equal quality of convergence rates. Hence, we consider norm-wise cancelation bits to determine the mantissa width for step 2 in AIR.

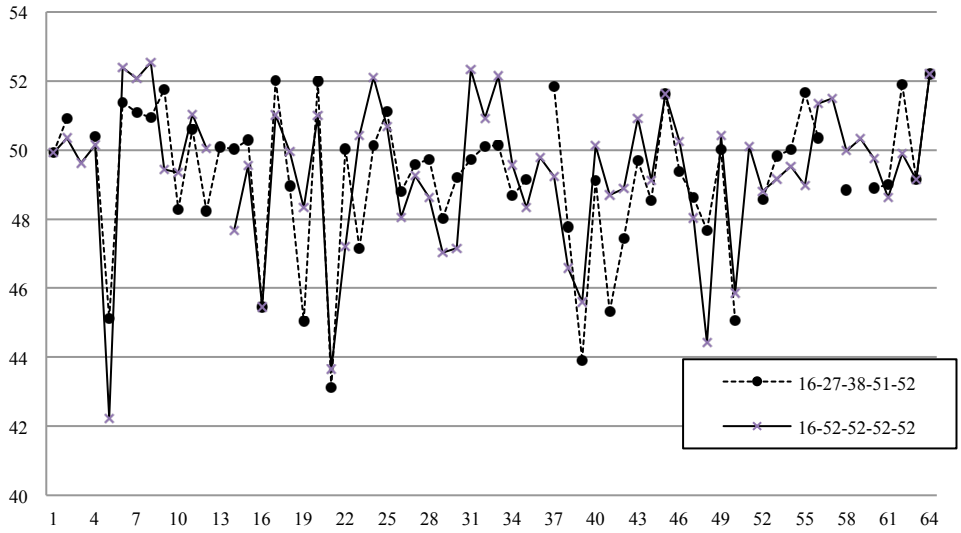

Figure 15. Convergence rate comparison

Now, we test 100 64 ×64 GRMs to examine theoretical accuracy and performance for AIR by comparing those from XMIR and BCIR of [19]. The BCIR of [19] still has limitation to be applied to real applications since it should predict the required number of iterations before computation, while AIR does not have the limitation. We employ 13 bits for the mantissa bit width for the lower precision for XMIR, BCIR, and AIR. BCIR is executed as increasing the order (i.e., required number of iterations) until it reaches the prescribed accuracy since the required number of iterations is not predictable before computation. Please refer to [19] for the algorithm of the BCIR. Intermediate bit widths between 13 and 52 bits are employed for AIR and BCIR for step 2. The golden results are obtained by direct method using double precision GEPP. We terminate the iteration of both AIR and XMIR when backward norm-wise error is smaller than the one from double precision direct method. Since the double precision GEPP implies the well-behaved [34,

62], the termination of iteration proves the Theorem 5.2 and 5.3 (i.e., well-behaved condition implies numerically stable). Based on the experiments, all test matrices satisfy Theorem 5.2 and 5.3.

Next, we examine Theorem 5.4. We compare the numbers of iterations of AIR to XMIR. Figure 16 depicts the numbers of iterations for AIR and XMIR to obtain higher accuracies than the direct method. The x axis represents $\log_2$-based condition numbers and the y axis represents the required numbers of iterations. Figure 16 proves Theorem 5.3, if the system is not too ill-conditioned (e.g., $\kappa(A) < 2^{13}$), since AIR requires at most one more iteration than XMIR. When the condition numbers are large, AIR sometimes requires 3 more iterations than XMIR in the figure. In this case, the relative error in the solution in step 3 is not relatively smaller than unity (e.g., XMIR requires more than ten iterations).

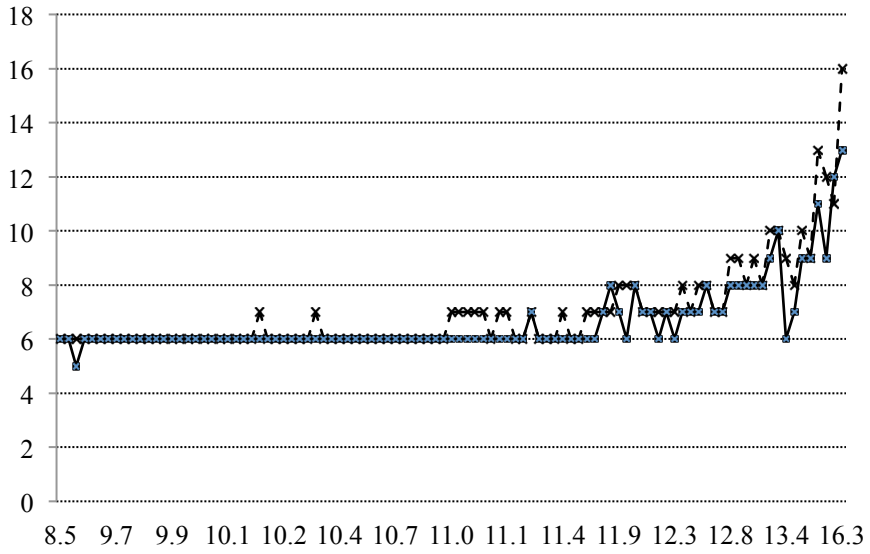

Figure 16. Numbers of iterations (AIR: dashed line, XMIR: solid line)

Next, we compare theoretical run times for AIR, XMIR, and BCIR to verify Theorem 5.5. The run time for BCIR is estimated assuming the required number of iterations is already known before computation. Figure 17 and 18 describe the relative run times for the BCIR, XMIR, and AIR compared to direct methods when the iterative refinements produce higher accuracy than the direct methods in terms of backward error for $\beta = \{1, 2\}$. Therefore, the run times for the direct methods are all 1s. The x axis represents condition numbers of matrices and y axis represents relative execution run times compared to the direct method.

AIR generally shows shortest run times regardless of condition numbers. The run times for AIR fluctuate due to dynamically increasing precisions while BCIR increases precisions statically and XMIR employs a static precision for all iterations. Based on the experiments, the max speedup of AIR over XMIR is around 1.2 X for $\beta = 1$ and 1.6 X for $\beta = 2$ when $\gamma = 0.25$. Therefore, the maximum speedup is bounded as $(\beta + 1)$ X, which corresponds to Theorem 5.4.

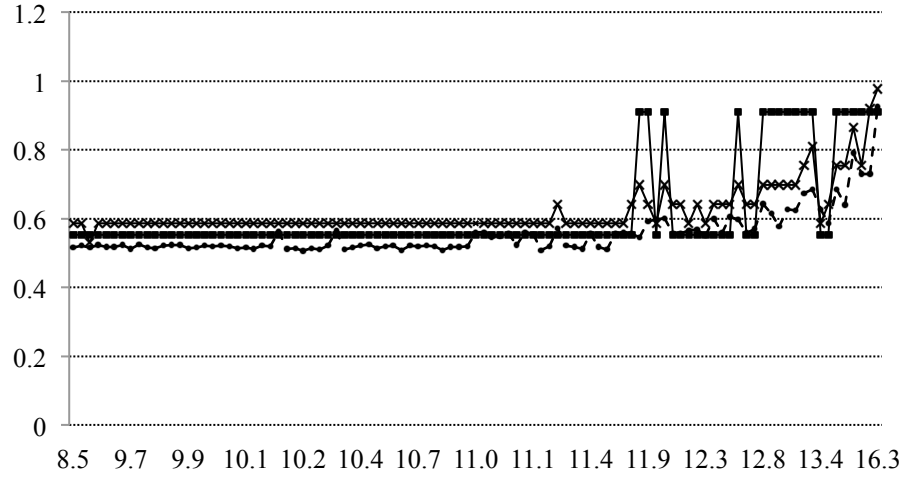

Figure 17. Run time when $\beta = 1$ (AIR: dashed, XMIR: solid ×, BCIR: solid ■)

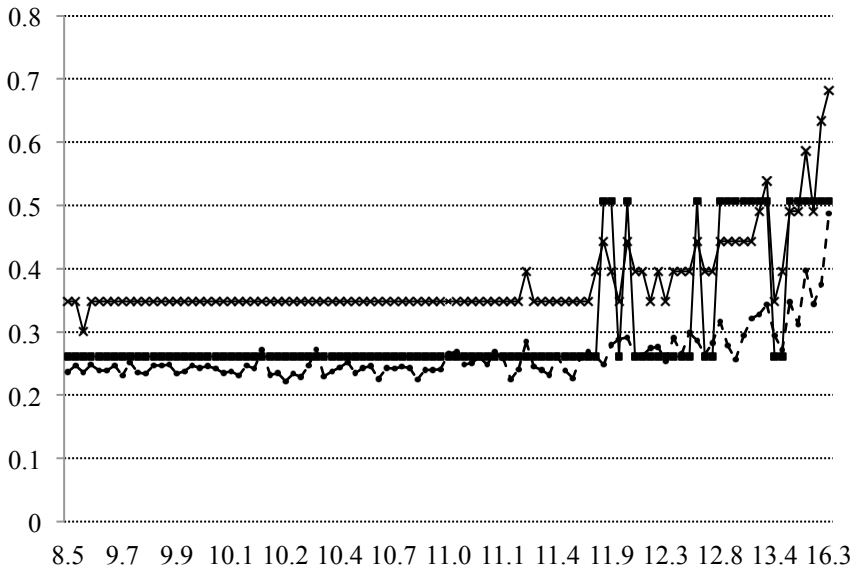

Figure 18. Run time when β = 2 (AIR: dashed, XMIR: solid ×, BCIR: solid ■)

Chapter 6

Implementation of adaptive dynamic precision iterative refinement

This chapter describes the implementations of steps 2 and 3 for AIR on the XUPV5-LX110T FPGA development board [25]. The description of the implementation of steps 1 and 4 is excluded in this chapter, since the implementation of step 4 is trivial and step 1 was previously explored in [9]. The numeric results for single precision implementation of steps 2 and 3 are verified along with an embedded Microblaze processor [82] by comparing the results to the ones from the MATLAB. The implementation employs VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) and Xilinx ISE/EDK 13.2. The programming interface for AIR is also discussed so that the implementation can be easily ported to a reconfigurable computing system [6, 22]. Since the implementation is parameterized with respect to precisions and pipeline depths of floating point arithmetic units, it can be easily adapted according to various precisions. The time-, clock-, and power-based performances of the implementations are discussed and compared to a newer FPGA Xilinx XC6VSX475T, since the XUPV5-LX110T [25] is an old FPGA.

6-1. Implementation on XUPV5-LX110T

This section discusses the implementation of steps 2 and 3 for AIR on the XUPV5-LX110T board. The implementation of step 2 is discussed at first and the implementation of step 3 later.

Seeking a residual (i.e., step 2) consists of dot products and subtractions. The implementation for step 2 employs a block method so that it can handle large matrices. A Processing Element (PE) performs a dot product in row wise direction as in Figure 19. The block size in row wise is set to 1024, but it can be changed according to the user preference.

To illustrate the operations for step 2 on the FPGA, we exemplify the case in which the block size is 1024 and the number of PEs is 32. Figure 19 describes the step 2 operation on the FPGA when the matrix size is 2K. Hence, the number of sub blocks in row wise is 2 and the number of sub blocks in column wise is 2048/32 = 64. Initially, the elements of the vector *b* are sent to the PEs accordingly. For example, the first element of the residual from the first PE is represented by $(b_0 - A_{0,0\text{-}1023}\, x_{0\text{-}1023}) - A_{0,1024\text{-}2047}\, x_{1024\text{-}2047}$. The b_0 is sent to the FPGA before activating the operations on the FPGA and the partial result from the block $(b_0 - A_{0,0\text{-}1023}\, x_{0\text{-}1023})$ is stored to a temporary register. Next, the partial result is used for subtraction at the dot product for the second block. Due to row-wise dot product operations, 32 elements of the residual can be calculated at the same time (i.e., $(b_{0\text{-}31} - A_{0\text{-}31,0\text{-}1023}\, x_{0\text{-}1023}) - A_{0\text{-}31,1024\text{-}2047}\, x_{1024\text{-}2047}$). Next 32 PEs calculate the next 32 residual elements using the process described above. A host generates a software reset signal to send the elements of the vector b before activating the process for the next dot products on the FPGA. The procedure relatedly is performed until the FPGA produces the last element of the residual.

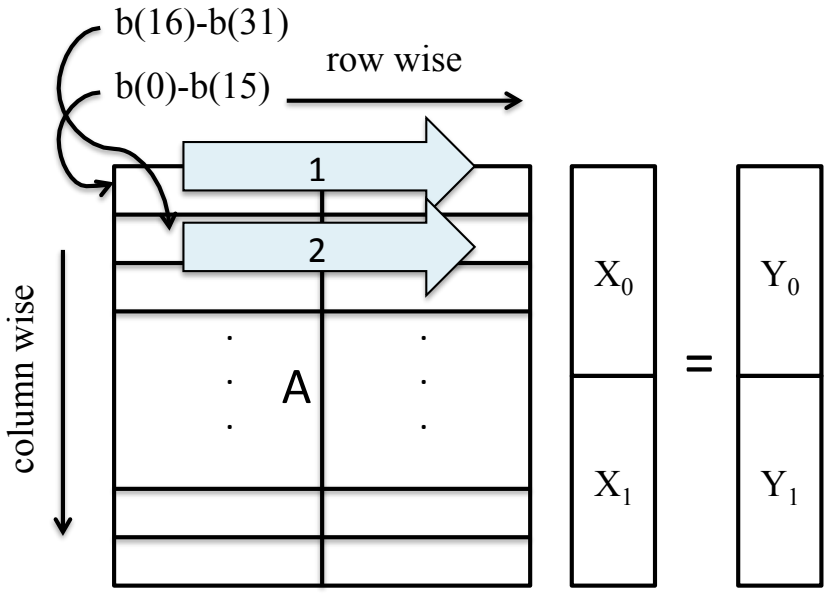

Figure 19. Procedure for step 2

Figure 20 describes the programming interface for the FPGA implementation for step 2. The following procedures are required for the programming interface. The procedures for the programing interface are :

1. Load the parameters such as the required number of row wise blocks and the block size to the FPGA so that the step 2 can be performed according to the parameters. For example, a PE is designed to perform a dot product according to the block size (e.g., 1K) and the required number of blocks for the dot product (e.g., 2 in row wise when the matrix size is 2K). The required number of column wise blocks is calculated using the number of PEs and the matrix size on host side.

2. Load partial elements of the matrix A into block memories inside PEs accordingly before initiating operations. Load the elements of approximate solution

vector into a block memory outside PEs and an element of the vector b into the partial sum register in each PE accordingly.

3. Set the first block memory status to be full and the second block memory status to be empty.

4. Activate operations in FPGAs by giving an enable signal.

5. Watch the status of FPGAs in the host to detect the completion of the dot block for the assigned PEs.

6. When the process 5 is done, initialize the FPGAs for the next dot products. For example, nullify the contents in the two buffer status registers indicating the buffers are empty with data to prevent the FPGA from activating. The results (i.e., elements of the residual) can be saved in some memories. We exclude this part for the implementation, but the implementation supports the output port for each PE so that users can save the elements of the residual according to their preference.

7. Perform the processes of 2 and 3 for dot products for next column block; We do not need to perform the procedure 4 since setting the first buffer to be full can activate the FPGAs.

8. Perform the processes of 6 and 7 until the dot product in the last row of the matrix is complete.

For the procedure 1 and 2, the Microblaze sends the number of row wise blocks, the block size, the elements of matrix A, and the approximation solution x to the FPGA before execution of step 2. We employ two software interface 32 bit registers to control

data transfer between the module of step 2 and the Microblaze. One register is used to indicate a PE ID so that only the indicated PE is ready to receive the data of corresponding matrix elements and vector elements. We map each bit of the register to each PE so that this 32 bit register can indicate 31 PEs and the BRAM storing *x*. The other register is used to indicate one type of data out of the matrix size *n*, an element of the vector *b*, block size, the number of PEs, matrix elements, and approximation solution vector elements in order to send them to the indicated PE accordingly. The synchronization for data transfer between the Microblaze and the module of step 2 should be considered. For the synchronization, we employ *while loops* to make sure the data transfers complete. Aforementioned procedures for the programming interface are implemented in C using API commands.

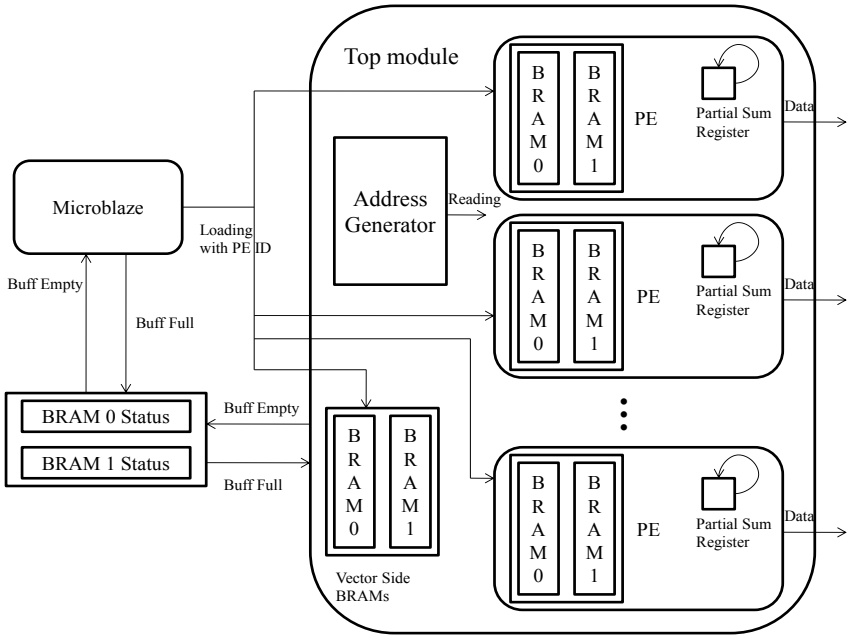

Figure 20. Multiple processing elements for step 2

Now, we discuss the hardware implementation for the programming interface. The XUP platform can emulate a reconfigurable computing platform, but the main difference in terms of programming interface is that the XUP does not support an API command to download bit stream files, while a reconfigurable computing such as Cray XD1 supports an API to download bit stream files. Xilinx EDK provides a template (i.e., *user_logic.vhd*) for the programming interface and the template provides programming interface registers, which can be accessible from both Microblaze and the module of step 2. In the template, two registers (e.g., *BRAM 0/1 Status* in Figure 20) are used to indicate buffer status. For example, the procedure 3 writes '1' to *BRAM 0 Status* register in Figure 20 so that the FPGA can consume the data in the block memory. When the FPGA consumes data completely in the block memory, the FPGA writes '0' to the *BRAM 0 Status* register so that the Microblaze can fill up the data into the block memory. Writing the elements of the matrix *A* and the vector *b* accompanies two types of signals. One signal indicates which PE is assigned and the other enables writing. The elements of the vector *b* are sent to the FPGAs only once per column block while the elements of the matrix *A* are sent to the FPGAs as many as the number of row wise blocks per column block. Double buffering is considered for the data transfer from the Microblaze to the module of step 2 to hide some latency for the data transfer. A BRAM can hold 1,024 elements in a vector for the current design. Initially, the matrix and vector data are fed into BRAM0. The BRAM0 status registers are then set to '1', representing the BRAM is ready for the operations. When the FPGAs completely consume the data from BRAM0,

the FPGAs reset the BRAM0 status register so that the Microblaze can load the data into the BRAM0 again. While the Microblaze loads data into BRAM0, the PE performs the dot product using the data in BRAM1. Consequently, this method can help hide latency for the data transfer from the Microblaze to the module of step 2 to some extent.

The *Partial Sum Register* in Figure 20 initially contains an element of b, partial results of dot products later, and finally, contains an element of the residual. The *Address Generator* in Figure 20 is used to read the data from BRAMs.

One floating point multiplier and adder are employed in single PE to perform a dot product and subtraction as shown in Figure 21 to save hardware resources and employ multiple PEs effectively. We employ Xilinx floating point IP cores generated by Xilinx Coregen 13.2. Each PE has two dual port BRAMs and one partial sum register. Once the pipeline of the adder is full with data, the reduction operation is initiated by feeding the output back to the input [33]. When the last product from the multiplier is sent to the adder, partial sums are stored into a register file for further reduction to produce a scalar value. The register file size depends on the pipeline depth of the adder. The run time for the reduction depends on the adder pipeline depth and it is negligible when the vector size is relatively large compared to the pipeline depth. A PE produces an element of a residual $r_j = b_j - (Ax)_j$ when the reduction produces a scalar value.

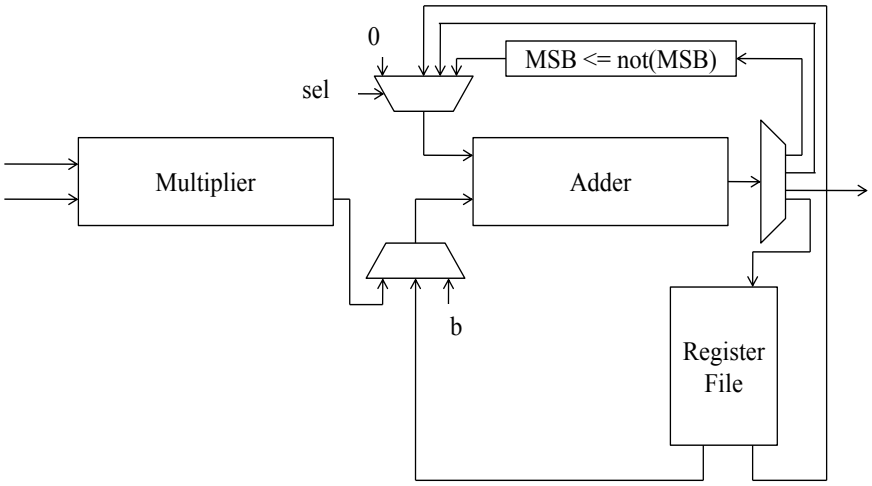

Figure 21. Single processing element for step 2

The entity of single PE is described in Figure 22. It consists of 17 types of input ports and 5 types of output ports. The port *clk* takes clock signals, *reset_n* takes reset signals, *cs* takes chip enable signal, and *rnw* takes read not write signal. The port *cs* and *rnw* can be used together to receive data from Microblaze as follows :

write_enable <= (cs) and (not rnw).

Microblaze generates the *write_enable* signal with data so that the BRAMs in the module of step 2 can store the data. A reset signal initializes variables and either the FPGA or Microblaze can generate the reset signal in order to perform the operations accordingly per column block. The *write_enable* accompanies the four types of input signals such as data signals (e.g., *data0* for matrix and *datax* for approximation solution in the entity), signals indicating addresses for writing (e.g., *addra* in the entity) and for reading (e.g., *raddr* in the entity), signals to indicate which PE in a column block ready for writing

(e.g., *pe_id* in the entity), and signals to indicate which BRAM out of the two in a PE ready for writing (e.g., *wen0* to indicate *BRAM0* and *wen1* to indicate *BRAM1* for writing in Figure 20). The *n_subblocks* in the entity represents the number of the row wise blocks so that the FPGA repeatedly performs the dot product operation in a row as many as the number of the row wise blocks. Once the dot product in a row is complete (i.e., the dot products for a column block are complete), the Microblaze generates a software reset signal so that FPGAs are ready to perform dot products for the next column block. The two signals, *buff0_rdy_in* and *buff1_rdy_in* are used to read the contents of *BRAM0/1 Stauts* registers in Figure 20 so that the module of step 2 can see if the data in the BRAMs are ready to be used. The *cur_buffer* signal is set to '0' indicating *BRAM0* as the current BRAM to be accessed when a reset (e.g., *reset_n* in the entity) signal is applied. The value in the *cur_buffer* signal is toggled when the FPGAs completes the dot product in a row wise block. The *cur_buffer* signal is used to choose one of the data read from the two BRAMs for the matrix A and vector x. The *buffer_ready* signal is used to activate the *Address Generator* in Figure 20. Finally, an element of the residual is transferred through *dataout* signal along with the *valid* signal, so that the Microblaze recognizes the completion of the dot product of a row (i.e., all the dot products in a column block are complete at the same time on FPGAs).

```
entity pes is
   port (clk : in  STD_LOGIC;
         reset_n: in std_logic;
         pe_sel: in std_logic;
         en: in std_logic;
         data0: in std_logic_vector(precision-1 downto 0);
         datax: in std_logic_vector(precision-1 downto 0);
         n_subblocks: in std_logic_vector(31 downto 0);
         matsize: in std_logic_vector(31 downto 0);
         b : in std_logic_vector(precision-1 downto 0);
         rnw: in std_logic; -- interface with the board
         cs: in std_logic; -- interface with the board
         wen0: in std_logic; -- initial loading to buff0
         wen1: in std_logic; -- initial loading to buff1
         addra : in std_logic_vector(31 downto 0);
         raddr: in std_logic_vector(blocksize-1 downto 0);
         dataout: out std_logic_vector(31 downto 0);
         buff0_rdy_in: in std_logic_vector(31 downto 0);
         buff1_rdy_in: in std_logic_vector(31 downto 0);
         cur_buffer: out std_logic;
         buff_ready: out std_logic;
         buff_change: out std_logic;
         valid: out std_logic );
end pes;
```

Figure 22. VHDL entity of single processing element for step 2

A PE to solve 64 × 64 block triangular matrices is implemented on the FPGA to perform step 3. FPGA implementation to solve a triangular system discussed in [83] did not consider division operations, since the implementation focused on solving triangular systems for LDL^T decomposition for Cholesky factorization. The implementation in this section considers the division to solve triangular matrices and the latencies from the divisions are hidden in the implementation. We first illustrate the operations for step 3 on the FPGAs when a triangular matrix size is 512 as in Figure 23. Solving the triangular matrix using the PE on the FPGAs requires the following steps :

SBT-1. (D-3) Solve the first 64 × 64 block triangular matrix: $L_{11}\, y_1 = z_1$; Find y_1.

SBT-2. (D-2) Update z_2 using the module of step 2: $z_2 = z_2 - L_{21}y_1$,

$z_3 = z_3 - L_{31}y_1$, and $z_4 = z_4 - L_{41}y_1$; Each $z_{1,2,3,4}$ has 64 elements.

SBT-3. (D-3) Solve the second 64×64 block triangular matrix: $L_{22}y_2 = z_2$; Find y_2.

SBT-4. Repeat the process from SBT-1 to 3 until y has 256 elements; Find y_4.

SBT-5. (D-2) Update z vectors using the implementation of step 2:

$z_{5\text{-}8} = z_{5\text{-}8} - L_{51\text{-}84}\, y_{1\text{-}4.}$

SBT-6. Repeat the process from SBT-1 to 4 until y_8 is found.

SBT stands for Steps to solve Block Triangular Systems. In the steps, (D-2) requires the Microblaze to download the bitstream file of the module of step 2 and (D-3) requires the Microblaze to download the bitstream file of the block triangular matrix solver module. Solving a large triangular matrix may require the host to download bitstream files multiple times.

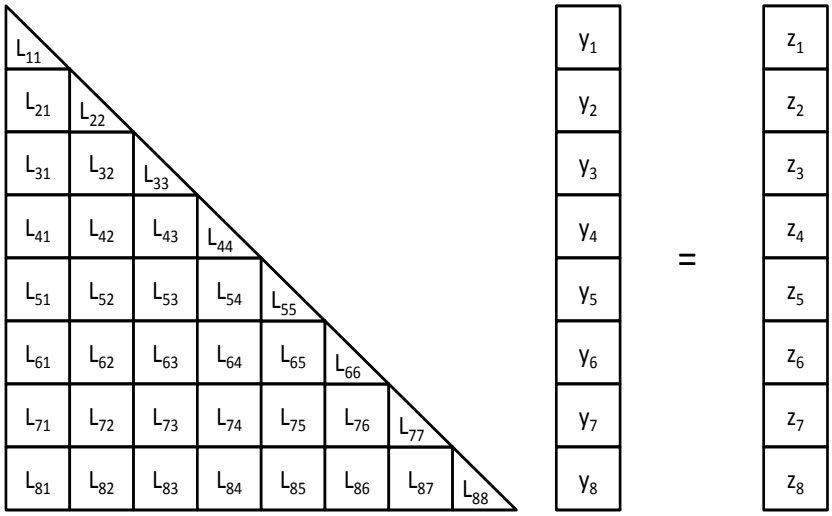

Figure 23. Block method for step 3

Multiple block sizes can be considered for effective data transfer according to hardware resources of a FPGA. For examples, the dot products of 64 elements are required to update $z_{2\text{-}4}$, but the dot products of 256 elements are required to update $z_{5\text{-}8}$. Since the block L matrices having different indices (e.g., L_{ij}, where $i \neq j$) are square matrices, employing the design of step 2 produces approximately 2 floating point operations per PE per clock cycle. Hence, when the square matrix size is relatively large compared to the block triangular matrix size, it can produce almost similar performance with step 2 since the operations using the module of step 2 dominate computation for step 3 on the FPGA. The required number to solve a block triangular matrix is determined by dividing a matrix size by the size of the block triangular matrix (e.g, 8 times are required for solving block a triangular matrix when the matrix size is 512 and the size of the block matrix is 64).

The implementation for the Block Triangular Matrix Solver (BTMS) consists of the five sub-modules as described in Figure 24. The *PE* module employs one multiplier, one adder, and two types of BRAMs to update z vectors. The *Arbiter* takes pulse signals (i.e., *zf_done*) from the *PE* to initiate divisions to produce solutions. The *zf_done* pulse signal is set when the first element of *z* has been found so that the division process can be initiated. A pulse signal is used for the *zf_done* signal instead of a steady high signal to recognize next events. To solve a lower triangular matrix, the diagonal elements of the matrix can be stored as '1'. When the *Arbiter* takes the *zf_done* pulse, it reads a diagonal element of the matrix from *Td BRAM* to divide. The dividend from *PE* and the divisor from *Td BRAM* are sent to *Div_inter* along with a pulse signal to activate the division to produce an element of the solution. Once the division is complete, it produces an element of solution along with a complete pulse signal to indicate the completion to *PE* and *XADDR* module. Once the *XADDR* module recognizes the completion of the division from *Div_inter* module, the *XADDR* generates a corresponding address to store the data. Whenever the *XADDR* takes a pulse signal (e.g., *Xdone*), it increments the address value with one. The address is sent to *Td BRAM* module along with the value obtained from *Div_inter* in order to store it at the corresponding address. The *Td BRAM* is a dual port RAM and it initially stores diagonal elements of a triangular matrix and later contains the solution.

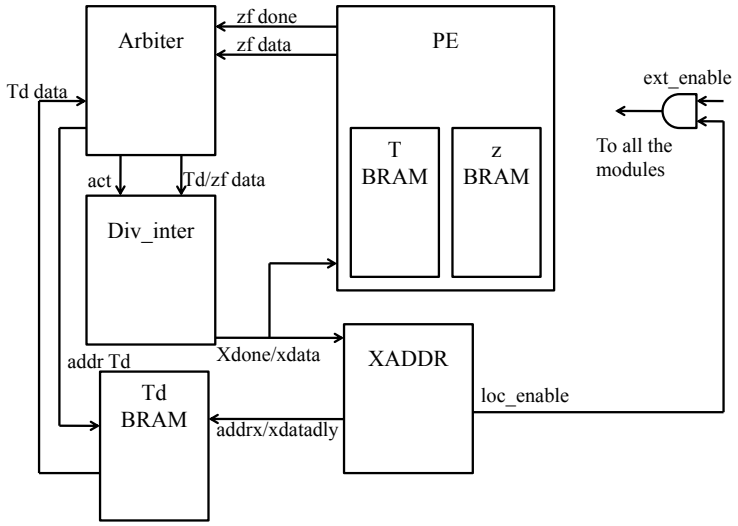

Figure 24. FPGA implementation for triangular system solver

The *XADDR* provides a local enable signal to cease the operations on the FPGAs when all the elements of the solution in the block are found. For example, when the reset signal is applied, the local enable signal is set to high again indicating the operation on the FPGAs is ready for the next operations. When the external enable signal is on, the *XADDR* starts incrementing addresses whenever *Xdone* is applied. When the address indicates the last address of a vector in the block, the local enable signal is set to low to stop the operation on the FPGAs so that unnecessary additional operations do not ruin the contents in the *Td_BRAM*. Once x_0 is found, the procedure for finding z_1, z_2, ..., and z_{63} can be initiated. The PE contains two types of BRAMs. One holds the block triangular matrix and the other stores intermediate results (i.e., z vectors). Since the implementation contains one multiplier and one subtractor, two operations can be performed per clock cycle. Diagonal elements are separately stored outside the *PE* to perform the divisions

effectively on the FPGA. When the division is complete, the *Xdone* pulse signal is applied to the *PE* module to allow the *PE* to be initiated to seek new intermediate *z* elements. However, when the solution element is found, but the previous process for seeking intermediate *z* elements is not complete yet, the *PE* waits until all the *z* elements are found before it begins the new process to find the new *z* elements.

To determine a precision at the next iteration in AIR, the required number of bits at the next iteration is explored. The number of required bits can be represented using the convergence rate σ as follows :

$|| r^{(1)}||_\infty = \sigma^{(1)} \times || r^{(0)}||_\infty$, where $\sigma^{(1)}$ is a convergence rate at the first iteration.

The infinity norm of a residual can be represented as follows :

$|| r^{(1)}||_\infty = \max_i | r^{(1)}_i |$, where $| r^{(1)}_i |$ is an absolute value of each component *i* in the residual vector $r^{(1)}$. Based on IEEE 754 floating point data representation, the infinity norm of a residual can be represented using an exponent value as follows :

$|| r^{(0)}||_\infty = 1.xxxx \times \exp^{(0)}$, where the data of the residual is in the range $\exp^{(0)} \leq ||r^{(0)}||_\infty < 2 \times \exp^{(0)}$. Likewise, $\exp^{(1)} \leq || r^{(1)}||_\infty < 2 \times \exp^{(1)}$. The exponents can be represented using the actual magnitude $\tilde{\gamma}$ in the register as follows: $\log_2 \exp^{(0)} \coloneqq \tilde{\gamma}^{(0)}$ and $\log_2 \exp^{(1)} \coloneqq \tilde{\gamma}^{(1)}$. The possible minimum value of the convergence rate is:

$$\min (\sigma^{(1)}) = \min (||r^{(1)}||_\infty / ||r^{(0)}||_\infty) = \min (||r^{(1)}||_\infty) / \max (||r^{(0)}||_\infty) > \exp^{(1)}/(2 \times \exp^{(0)}).$$

Hence, $\min(\sigma^{(1)}) > \exp^{(1)}/(2 \times \exp^{(0)})$. The possible maximum value of the convergence rate is:

$\max(\sigma^{(1)}) = \max(\|r^{(1)}\|_\infty / \|r^{(0)}\|_\infty) = \max(\|r^{(1)}\|_\infty) / \min(\|r^{(0)}\|_\infty) < 2\exp^{(1)}/\exp^{(0)}$. Hence, $\max(\sigma^{(1)}) < 2\exp^{(1)}/\exp^{(0)}$. Therefore, $\exp^{(1)}/(2\times\exp^{(0)}) < \sigma^{(1)} < 2\exp^{(1)}/\exp^{(0)}$. The required number of bits can be represented using $-\log_2$ operator as follows :

$$\tilde{\gamma}^{(0)} - \tilde{\gamma}^{(1)} - 1 = -\log_2(2\exp^{(1)}/\exp^{(0)}) < -\log_2 \sigma^{(1)} < -\log_2(\exp^{(1)}/(2\times\exp^{(0)})) = \tilde{\gamma}^{(0)} - \tilde{\gamma}^{(1)} + 1.$$

Therefore, the required number of bits between two consecutive residuals in AIR can be represented as : $-\log_2\sigma^{(1)} = \tilde{\gamma}^{(0)} - \tilde{\gamma}^{(1)} + 1$. (6.1)

Based on (6.1), AIR in Algorithm III is revised to Algorithm III-A for the implementation.

Algorithm III-A. Adaptive dynamic precision iterative refinement :

The adaptively chosen mantissa widths for steps 2 and 4 :

$t^{(1)} = 2\ t_L$ if $t_A > 2\ t_0$, or $t_1 = t_A$ if $t_A \le 2\ t_0$,

$t^{(2)} = t_L + 2 \times (\text{Max}\ |\exp_b| - \text{Max}\ |\exp_r^{(1)}| + 1)$,

$t^{(3)} = t_L + (\text{Max}\ |\exp_b| - \text{Max}\ |\exp_r^{(2)}|) + (\text{Max}\ |\exp_r^{(1)}| - \text{Max}\ |\exp_r^{(2)}|) + 2$,

$t^{(i)} = t_L + (\text{Max}\ |\exp_b| - \text{Max}\ |\exp_r^{(i-1)}|) + (\text{Max}\ |\exp_r^{(i-1)}| - \text{Max}\ |\exp_r^{(i-2)}|) + 2$.

if $t^{(i)} > t_A$, $t^{(i)} = t_A$.

Note.

$t^{(i)}$: the required mantissa bits at the i^{th} iteration
t_L : the required mantissa bits for the lower precision
t_A : the original precision

The Microblaze decides the precision for the next iteration, since the operation required for the decision is O(n) which is negligible compared to the complexity $O(n^2)$ in step 3. Hence, before the completion of step 3 on the FPGA, the comparison among all

the elements of a residual can be processed in the Microblaze to determine the precision for the next iteration. The following C code example detects the maximum absolute component (i.e., *maxnorm* in the code) in the exponent part in a residual.

```
/* Infinity norm detection */
residus[0] =  0x7fffffff & residus[0]; //take absolute value of the first component
maxnorm = inf_norm[0]>>m_width; //take the exponent part only
for (i =1; i< n_pe; i++) // comparison
{ residus[i] =  0x7fffffff & residus[i];
compnorm = residus[i]>>m_width;
if(maxnorm<compnorm) maxnorm = compnorm;}.
```

Now, we discuss the programming interface for AIR on a reconfigurable computing platform based on the implementation on the XUP board. Even though each reconfigurable computing has its individual feature, most reconfigurable computing platforms support their flexible API commands. Hence, a user can adapt the implementation according to API commands supported by the reconfigurable computing platform. Let us assume that a user wants the original precision accuracy in backward error and a local folder contains separate multiple bit-stream files each for steps 1, 2 and 3 according to applied precisions. Based on the assumptions, the procedure to execute AIR on reconfigurable computing is as follows :

RC-1: The bit stream file for a lower precision LU decomposition is downloaded to generate the lower and upper triangular matrices. The lower and upper triangular matrices can be stored into local memory on the FPGAs or host memory. It is recommended to store them to a local memory to exploit better bandwidth as long as they can be fit into the local memory (e.g., Onchip memory on the FPGAs).

RC-2: Next, using the lower and upper triangular matrices, the bitstream file for the triangular system solver of the lower precision is downloaded to produce an approximate solution. The approximate solution can be stored on a host memory.

RC-3: The bit stream file for the step 2 of an arbitrary precision is downloaded to be executed and produce a residual. The residual is sent to a host memory to see if the numeric solution satisfies the termination criterion for iteration. Also, the maximum absolute exponent value of the residual elements is picked by the host in order to decide a precision for the next iteration.

RC-4: The bit stream file for triangular system solver is downloaded again to produce an approximated error from previous solution to correct the solution in next step.

RC-5: The bit stream file for addition of an arbitrary precision is downloaded to correct the solution.

RC-6: Go to RC-3 until a solution meets the termination criterion.

The decision for a precision at the next iteration is performed on host side while steps 3 and 4 are executed on the FPGAs, so that the latency for the decision in the host can be hidden.

6-2. Estimation for time-based performance

This section discusses the time-, clock-, and energy-based performances of the implementations of steps 2 and 3 on the FPGAs for various precisions. The performance estimation for the implementations for steps 2 and 3 on the FPGAs is based on the assumption that each PE in step 2 can perform 2 floating point operations per clock cycle and the achievable performance in step 2 is approximately equal to the achievable performance in step 3. The validation of the assumption is as follows :

Validation : Ignoring the data transfer from the host, the required number of clock cycles for step 2 for each row is represented as follows:

$$C_{FPGAs\text{-}ROW} = (n/BS) \times (BS + k + l + r) = n + (k + l + r)/BS$$

where *k* is the number of pipeline stages for the multiplier, *l* is the number of pipeline stages for the adder, *BS* is the block size, *n* is the matrix size and *r* is the number of cycles required for the reduction. Total run time is determined as follows:

$C_{FPGAs} = \{n + (k+l+r)/BS\} \times (n/n_{PE})$, where n_{PE} is the number of PEs.

If $n \gg (k + l + r)/BS$, then $C_{FPGAs} \approx n^2/n_{PE}$. (6.2)

Since the run time for step 3 mainly depends on dot products, we assume that the required number of clock cycles for step 3 is equivalent to step 2 on the FPGAs. However, notice that the degree of parallelism for the computation is different between the step 2 and 3. The required operations for step 2 can be fully parallelized, but the operations for step 3 can be partially parallelized. For example, if the block size of the

sub-triangular matrix is 64 and the second block size is 1024 to solve a triangular matrix of n = 2K, the possible parallelism can be represented as follows :

1 (i.e., solving triangular system) - 64 (i.e., matrix-vector multiplication) - 1 - 64 - … - 1 – 1024 - 1 - 64 - …, and so on. When a matrix size is 2K, the computation of step 3 requires 32 (= 2K/64) times to solve sub triangular matrices, 240 (=15×(15+1)) times for 64 × 64 matrix vector multiplications and 1 1024 × 1024 matrix vector multiplication. The total operations are $2(2K)^2 = 8(1024)^2$ to solve the triangular matrix. Out of $8(1024)^2$ operations, only 2048/64 = 32 block triangular matrix solvers are required and step 2 operations are employed for the other operations. Hence, in step 3 operation, $32 \cdot 2 \cdot 64^2 / (8(1024)^2) = 3\%$ are used for the block triangular solvers and 97% are used for step 2 implementation.

The Xilinx Place And Route (PAR) in ISE 13.2 is used to estimate the required hardware resources and achievable clock rates for the implementations of steps 2 and 3 on the FPGAs. Since the implementations of steps 2 and 3 were realized on the XUPV5-LX110T board (e.g., 65nm production process), the actual results on the board are discussed at first and compare them to a newer FPGA Xilinx XC6VSX475T (e.g., 40 nm production process). Finally, the estimated performances on the XC6VSX475T are compared to a new GPU Nvidia Tesla C2075 (e.g., 40 nm production process).

Based on Xilinx ISE PAR, Table VII shows the required hardware resources, allowable clock rate, and estimated performance for the implementation of step 2 and BTMS on the Xilinx XC5VLX110T FPGA in the XUPV5-LX110T board when single precision is applied.

TABLE VII. SINGLE PRECISION STEP 2 ON THE XILINX XC5VLX110T

Impl'	DSP 48E	Slices	# of BRAM (36Kb)	# of PEs	PAR CLK App' CLK	GFLOPs
Step 2	***59/64***	8,906/17,280	62/148	14	125.5 MHz 125 MHz	3.5
BTMS	***7/64***	1,789/17,280	38/148	1	125.6 MHz 125 MHz	0.25

The Microblaze was also considered to estimate hardware resources for both step 2 and BTMS. The implementation of step 2 on the Xilinx XCVLX110T FPGA contains 14 PEs. Numeric results from the implementation of step 2 and BTMS were verified by comparing the results with MATLAB (i.e., 2 ulps are different at most between the results from the FPGA and MATLAB for a dot product of 1024 elements). Single precision floating point arithmetic units employ DSP48s for high speed. Each slice in Xilinx Virtex-5 FPGA contains four LUTs and four flip-flops [84]. Each single precision PE contains 4 DSP 48Es in the implementation of step 2. The hardware resource limitation comes from the number of DSP48Es, limiting the number of PEs to 16 for this device without consideration of the Microblaze. BRAM is usually limited for the implementation of BTMS on the Xilinx XC5VLX110T (Notice that increasing twice for the size for the matrix requires four times more BRAM resources). The block size of triangular matrices can be increased in larger FPGAs.

Since the Xilinx XC5VLX110T FPGA is a smaller and older FPGA, a newer FPGA Xilinx XC6VSX475T is considered for the performance evaluation. The hardware resource limitation for step 2 comes from slices for the Xilinx XC6VSX475T. The DSP 48Es are employed only for single and double precision floating point adders, since

current customized precision floating point adder IP blocks in Xilinx Coregen employ slices except for single and double precision. Multipliers employ DSP48Es. Table VIII shows the required hardware resources and allowable clock rates for step 2 employing single PE on the Xilinx XC6VSX475T without considering Microblaze. Each slice in the Xilinx Virtex-6 FPGA contains four LUTs and eight registers [85]. Slices limit the design to employ 65-170 PEs according to precisions. The limited number of PEs is calculated based on around 75% of slice usage. However, achievable clock frequencies can be decreased according to the increase of the number of PEs.

Table IX shows the variation of the achievable clock rates according to the number of PEs for step 2. Table X shows achievable performances for step 2 employing the allowable numbers of PEs.

TABLE VIII. ALLOWABLE NUMBER OF PES FOR STEP 2 ON XILINX XC6VSX475T

Precision		Pipe' Depth +/×	DSP 48E	Slices 1st row: LUT 2nd row: Reg	# of BRAM (36Kb)	Allowable # of PEs	PAR CLK Syn' CLK
Exp Size	Mant' Size						
8	23(S)	11/8	4/2,016	1,270/595,200 ***1,264/297,600***	4/1,064	170	165 MHz 255 MHz
11	38	12/13	5/2,016	2,351/595,200 ***2,167/297,600***	6/1,064	106	192 MHz 251 MHz
11	52(D)	14/15	13/2,016	2,907/595,200 ***2,632/297,600***	8/1,064	83	206 MHz 248 MHz
15	63	13/22	16/2,016	3,799/595,200 ***3,561/297,600***	10/1,064	65	187 MHz 230 MHz

Since AIR is useful when the run times for steps 2 and 3 are not negligible compared to the run time for step 1 (i.e., when a user requires an unusually high precision accuracy compared to the precision applied for step 1), the run times for steps 2 and 3 are estimated. The run time for the refinement procedure per iteration on the Xilinx XC6VSX475T is compared to the one on the NVIDIA Tesla C2075 GPU. Since the NVidia Tesla C2075 supports either single or double precision in terms of hardware, we assume that the GPU produce single precision performance for lower precision than single precision and produces double precision performance for higher precision than single precision and less or equal to double precision.

TABLE IX. CLOCK FREQUENCY VARIATION ACCORDING TO THE NUMBER OF PES

# of PEs	DSP 48E	Slices	# of BRAM (36Kb)	PAR CLK	GFLOPs
170	680/2,016	67,294/74,400	342/1,064	80 MHz	27.2
150	600/2,016	63,328/74,400	302/1,064	85 MHz	25.5
125	500/2,016	55,019/74,400	252/1,064	85 MHz	21.3
100	400/2,016	45,889/74,400	202/1,064	93 MHz	18.6
75	300/2,016	35,330/74,400	152/1,064	101 MHz	15.2
50	200/2,016	24,444/74,400	102/1,064	117 MHz	11.7
25	100/2,016	11,903/74,400	52/1,064	144 MHz	7.2
1	4/2,016	497/74,400	4/1,064	165 MHz	0.3

TABLE X. PERFORMANCE OF STEP 2 ON XILINX XC6VSX475T

Precision		DSP 48E	Slices LUT Register	# of BRAM (36Kb)	# of PEs	PAR Syn' [MHz]	GFLOPs
Exp Size	Mant' Size						
8	23(S)	680/2,016	67,294/74,400	342/1,064	170	80 242	27.2
11	38	530/2,016	69,591/74,400	321/1,064	106	79 238	16.7
11	52(D)	1,079/2,016	66,945/74,400	336/1,064	83	82 237	13.6
15	63	1,040/2,016	68,619/74,400	330/1,064	65	78 221	10.1

We also assume that the performance of step 3 is comparable to the performance of step 2 for both the GPU and FPGA. Hence, we only consider the performance of step 2 for the comparison. For the FPGA, we ignore the overhead to load bit-stream files and the time for data transfer from the host. For the GPU, we employ matrix vector multiplication from Magma v0.2 [44] and exclude the data transfer time from host to GPU global memory to make the comparison fair.

The GPU produces 62 GFlops for single precision and 31 GFlops for double precision when the performance is saturated [86]. Finally, the run time per iteration for both the FPGA and the GPU can be expressed as follows:

$T = Flops_{step3}/P_{step3} + Flops_{step2}/P_{step2}$, where *Flops* represents the required number of floating operations for step 2 and 3 and *P* represents the performance metric described as the number of floating point operations per second. Since the required number of floating point operations are equal for steps 2 and 3 as $2n^2$,

$T = 2n^2\ (1/P_{step3} + 1/P_{step2})$, $4n^2/P = 2n^2\ (1/P_{step3} + 1/P_{step2})$,

$P = 2P_{step2}\ P_{step3}/(P_{step2} + P_{step3})$.

For the GPU,

$$P_{GPU} \approx 2(62 \times 31)/93 \approx 41\ GFLOPs, \quad (6.3)$$

For the FPGA,

$$P_{FPGA} \approx \{2(27.2 \cdot 16.7), 2(27.2 \cdot 13.6), 2(27.2 \cdot 10.1)\} / \{27.2+16.7,\ 27.2+13.6,\ 27.2+10.1\}$$

$$\approx \{20.7,\ 18.1,\ 14.7\ GFLOPs\} \quad (6.4)$$

where {x, y, z} represents the cases when the required mantissa bit widths are 38, 52, and 63 (refer to Table IX). The FPGA implementation is desirable when users require accuracy beyond double precision since the GPU cannot handle this case directly in hardware.

The run time for AIR depends on γ^{β} as described in (5.6). It is assumed that β is around 2 for the FPGA, a lower precision is single precision, and convergence rate is around 2^{-15} for the comparison. Hence, the iteration m = {1, 2, 3}. When m = 2,

$$8n^2/P = 2n^2\ (1/P_{step3} + 1/P_{step2})_{i=1} + 2n^2\ (1/P_{step3} + 1/P_{step2})_{i=2},$$

Hence, $P = 19.3\ GFlops$. Likewise, $P = 17.5\ GFlops$ when m = 3. Therefore, in AIR,

$P_{FPGA\text{-}AIR} = \{20.7,\ 19.3,\ 17.5\ GFLOPs\}$ and $Speedup = \{1X,\ 1.1X,\ 1.2X\}$ over XMIR. Notice that a precision lower than single precision can be applied for AIR for a better performance and the impact of AIR can be maximized when a user requires an extremely high accuracy in the solution.

6-3. Clock- and energy-based performance

Many computational science applications are computationally intensive causing huge power consumption. In high performance computing, it is a challenge to keep the

power budget low while keeping high performance. Power has emerged as a significant constraint to high performance computing. There have been many research efforts for time-based performance modeling [87, 88]. Along with time-based performance, energy-based performance becomes a significant performance metric in high performance computing applications [89, 90]. We disucss an energy-based performance, which represents the achievable number of floating point operations per joule in this section. In [49], the energy-based performance is reported for MPIR when step 1 dominantly determines the performance. This section discusses the energy-based performance for AIR when a user requires an unusually high precision accuracy. Hence, the energy-based performances for refinement procedure per iteration are compared between the Xilinx XC6VSX475T FPGA and NVIDIA Tesla GTX 2075 GPU.

Total power consumption in digital circuits is described as follows [91]:

$$U = S + D = S + \tilde{\alpha}\, C\, V^2 \times f \tag{6.5}$$

where U is total power consumption, S is static power consumption, D is dynamic power consumption, C is an effective capacitance, $\tilde{\alpha}$ is a constant representing the number of transistors participating in switching activity, V is an operational voltage, and f is an applied clock rate. Now, we define three different performance metrics as follows.

$P :=$ # of Flops/sec,

$F_{CLK} := P/f$ = (# of Flops/sec)/(# of clock-cycles/sec) = # of Flops/clock-cycle,

$F_{JLE} := P/U$ = (# of Flops/sec) / (joule/sec) = # of Flops/joule.

The dynamic power consumption is related to clock based performance as follows:

$$F_{JLE\_D} = P/D = P/(\tilde{\alpha}\, C\, V^2 \times f) = F_{CLK} / (\tilde{\alpha}\, C\, V^2), \tag{6.6}$$

$\tilde{\alpha} = (F_{CLK} / F_{JLE\_D}) / (C\ V^2)$.

F_{JLE} represents the achievable number of floating point operations per joule and $F_{JLE\text{-}D}$ represents the achievable number of floating point operations per joule when the static power is already sufficient to perform floating point operations. It is important that a user seeks to maximize F_{CLK} to save energy based on (6.6). The effective capacitance in (6.6) depends on numerous factors such as the sizes of transistors, the interconnections among transistors, and so on.

6-4. Design effectiveness for dynamic power consumption

Assuming that the effective capacitances and operational voltages for the GPU and the FPGA are similar in equation (6.6), the dynamic power consumption is mainly determined by the clock-based performance and $\tilde{\alpha}$. Based on (6.6), either making the clock-based performance higher or lowering $\tilde{\alpha}$ makes the energy-based performance better. Obtaining a higher clock-based performance and lowering $\tilde{\alpha}$ are closely related each other. For examples, the $\tilde{\alpha}$ can be approximately represented as follows :

$$\tilde{\alpha} = \#_{COMP} + \#_{SUP}, \quad (6.7)$$

where $\#_{COMP}$ represent the number of transistors for actual computation per clock cycle (i.e., necessary transistors) and $\#_{SUP}$ represents # of transistors for supporting for computation per clock cycle (i.e., unnecessary transistors). Hence, lowering $\tilde{\alpha}$ means lowering the portion of # of transistors for supporting for computation, since # of transistors for actual computation per clock cycle is not reducible in a design. The F_{CLK} represents the number of actual Flops per clock cycle and it does not reflect the number

of Flops from supporting logic. For example, suppose 10 % of transistors are participating in the actual computation to produce floating operations and F_{CLK} is 10 Flops/cc in Design 1. In Design 2, suppose 50 % of transistors are participating in the actual computation to produce floating operations and F_{CLK} is 10 Flops/cc. If the two design employs the same FPGA (i.e., the same hardware resources), the F_{CLK}/α can be represented as follows :

In Design 1, $\#_{COMP1} = 0.1\ (\#_{COMP1} + \#_{SUP1})$ and in Design 2, $\#_{COMP2} = 0.5\ (\#_{COMP2} + \#_{SUP2})$. Since $\#_{COMP1} = \#_{COMP2}$ if we assume the necessary minimum numbers of transistors for computation are identical in an application, $0.1\ (\#_{COMP1} + \#_{SUP1}) = 0.5\ (\#_{COMP2} + \#_{SUP2})$.

$$F_{CLK1}/\tilde{\alpha}_1 = 10/\ (\#_{COMP1} + \#_{SUP1}),$$

$$F_{CLK2}/\tilde{\alpha}_2 = 10/\ (\#_{COMP2} + \#_{SUP2}) = 0.2\ F_{CLK1}/\tilde{\alpha}_1.$$

Hence, this example shows that reducing supporting logic can save dynamic power consumption 5 times since it can make $\#_{SUP}$ lower. More specifically, equipping more PEs on the FPGA can increase F_{CLK}, but $\tilde{\alpha}$ also increases according to the increase of hardware resources. Now, we define a design effectiveness coefficient Λ as follows : $\Lambda = \#_{COMP}/(\#_{COMP} + \#_{SUP})$, $0 < \Lambda \leq 1$. The design effectiveness coefficient represents how many percentage of logic participating in switching activity produce the actual computation. Since toggle rates for gates are different, we consider $\#_{COMP}$ and $\#_{SUP}$ as the converted number of transistors if the toggle rates are 100 % rather than the actual number of transistor. To state more clearly design effectiveness, we define another performance metric F_{JLE_D} / F_{CLK}. The design effectiveness coefficient can be represented using dynamic power consumption as follows :

$\Lambda = \#_{COMP}/(\#_{COMP} + \#_{SUP}) = \#_{COMP}/\tilde{\alpha} = C\ V^2\ \#_{COMP}\ F_{JLE\text{-}D} / F_{CLK} = C\ V^2\ \#_{COMP}\ f/D.$

Hence, given a frequency, lower dynamic power consumption means higher effectiveness of the design. The ratio of the two Λs for the two designs for an application can be represented using energy-based performances considering dynamic power consumption as follows :

$F_{JLE\_D1} = F_{CLK1} / \tilde{\alpha}_1\ C_1\ V_1^2,$

$F_{JLE\_D2} = F_{CLK2} / \tilde{\alpha}_2\ C_2\ V_2^2,$

Since $\tilde{\alpha}\ \Lambda = \#_{COMP}$ and $F_{CLK}/\tilde{\alpha} = \Lambda\ F_{CLK}/\#_{COMP}$,

$F_{JLE\_D1}/F_{JLE\_D2} = \Lambda_1\ F_{CLK1}/(\#_{COMP1} C_1\ V_1^2) / \Lambda_2\ F_{CLK2}/(\#_{COMP2}\ C_2\ V_2^2).$

Assuming that $\#_{COMP2}/\#_{COMP1} = F_{CLK2}/F_{CLK1}$ if the two designs employ the same application and the same production process for transistors,

$F_{JLE\_D1}/F_{JLE\_D2} = (\Lambda_1/\Lambda_2)\ (C_2/C_1)\ (V_2/V_1)^2,$ (6.8)

$\Lambda_1/\Lambda_2 = (F_{JLE\_D1}/F_{JLE\_D2})\ (C_1/C_2)\ (V_1/V_2)^2.$ (6.9)

If the two computing platforms are identical, $\Lambda_1/\Lambda_2 = \#_{COMP2}/\#_{COMP1} = (F_{CLK1}/F_{CLK2})$, since $\#_{COMP1} + \#_{SUP1} = \#_{COMP2} + \#_{SUP2}$. Based on (6.8), it is important for a user to produce higher Λ (i.e., lowering $\tilde{\alpha}$) to save dynamic power consumption. The Λ_1/Λ_2 in (6.9) expresses the effectiveness of design in terms of energy efficiency. For examples, the F_{CLK} can be higher on the FPGAs if the circuits are designed effectively given the same hardware resources (i.e., if the design obtains a higher Λ given an α). Therefore, given hardware resources, it is important for a designer to implement applications to achieve a higher performance per clock cycle in order to save power. In Von-Neuman architectures (e.g., micro-processors or GPUs), it is necessary to fetch data from instruction and data

memory and decode them. This frequent data access may require substantial part of power consumption in the application, which can increase $\#_{SUP}$ in equation (6.7). Especially, GPUs require large power consumption for the data access from memory for many applications [92]. Meanwhile, data driven architecture [20] such as FPGAs does not require that much data access, which can save substantial amount of power by reducing $\#_{SUP}$.

6-5. Estimation of clock-, energy-based performance and design effectiveness

The clock-based performance is estimated by dividing the time-based performances by corresponding clock rates based on (6.3) for the GPU and (6.4) for the FPGA. Since the GPU employs 1.15 GHz for core-clock [93], F_{CLK_C2075} = 35.7 Flops/cc for steps 2 and 3 (step 2 employs double precision and step 3 employs single precision). The FPGA employs different clock rates for steps 2 and 3. Hence,

$F_{CLK\_XC6\ M=38} = (F_{CLK\_step2} + F_{CLK\_step3})/2 = (27.2/0.08 + 16.7/0.079)/2 = 275.7$ Flops/cc. Likewise, $F_{CLK\_XC6\ M=52}$ = 234.7 Flops/cc. For single precision, $F_{CLK\_XC6\ M=23}$ = 340.0 Flops/cc. In AIR, $F_{CLK\_XC6\_AIR\ M=38}$ = 292.2 Flops/cc and $F_{CLK\_XC6\_AIR\ M=52}$ = 282.2 Flops/cc. Hence, the FPGA implementation in MPIR achieves up to 7 times more Flops/cc than the GPU in MPIR. The clock-based performance in AIR on the FPGA can achieve up to 8 times more Flops/cc than the GPU. The clock-based performance tells how many *actual* operations can be done given a clock cycle. If there exists unnecessary work on the computation, the clock-based performance becomes lower. However, it is dangerous to believe that higher clock-based performance means more power efficiency,

since we need to consider the design effectiveness coefficient Λ along with the clock-based performance to evaluate power efficiency. For examples, if a silicon die size is huge, it is obvious to obtain a higher F_{CLK} by exploiting the considerable quantity of transistors. Along with the Λ, the power efficiency can be evaluated since Λ represents how many portion of the silicon die can be used to produce actual floating point operations out of the transistors participating in switching activity. If some transistors do not participate in switching activity, they do not consume the dynamic power. Hence, when some design is downloaded on FPGAs, most transistors do not consume the dynamic power since they mostly participate in routing rather than switching activity [94].

The power consumptions for the execution for the step 2 on the XUPV5-LX110T board are physically measured using Wattsup? Pro [95] whose accuracy is 0.1 W. Before measuring the power consumptions for step 2, we need to handle dipswitches in the board [96]. Wattsup? Pro measures an instant power consumption every second. Figure 25 describes the measured instant power consumptions when the step 2 are run on the XUPV5-LX110T board when the numbers of PEs are 2, 6, 10, and 14 and the clock frequency is fixed at 125 MHz. This experiment explores the total power variation according to $\tilde{\alpha}$ variation. In Figure 25, the oscillations below 7.6 W represent the static power consumption approximately including a few of dynamic power consumption due to clock circuit itself.

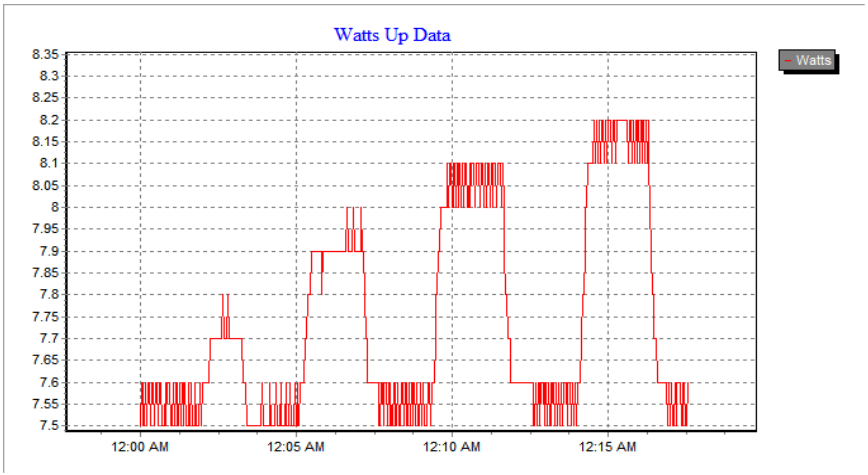

Figure 25. Power consumption measurement on the XUPV5-LX110T for step 2

The Xilinx Power Analyzer (XPA) is designed to estimate on-chip power consumption while Wattsup? Pro measures the power consumption on the entire board of XUPV5-LX110T. The XPA provides two options to estimate power consumption; TYPICAL and MAXIMUM. The XPA requires a Native Circuit Description (NCD), a Physical Constraint File (PCF), and Switching Activating Interchange Format (SAIF) files for accurate analysis [97, 98]. We employ the NCD and PCF files obtained from Xilinx ISE/EDK. Instead of SAIF, we employ the toggling rates set by default in the XPA except flip-flops, I/O, and DSP. The Xilinx Power Estimator (XPE) user guide suggests 50% for the toggle rates for flip-flops, I/O, and DSP for the step 2 (i.e., multiply-accumulation operation) [98]. Hence, we take the suggestion for the power consumption estimation. We compare the power consumption estimation in the XPA to the actual results measured by Wattsup? Pro. Table XI shows the power consumption variation according to the numbers of PEs for step 2.

TABLE XI. POWER MEASUREMENT ACCORDING TO NUMBER OF PES

# of PEs	Wattsup? Pro (W) XUPV5-LX110T board	XPA (W) XC5VLX110T chip -Typical-	XPA (W) XC5VLX110T chip -Maximum-
2	7.706	1.262	1.905
6	7.882	1.407	2.082
10	8.031	1.569	2.253
14	8.163	1.695	2.387

Figure 26 shows the skew the power consumption variations based on the Table XI. Based on Figure 26, the dynamic consumption variations by the XPA and the Wattsup? Pro are similar each other, but the TYPICAL setting seems to be a little bit lower compared to the measured dynamic power consumptions. Hence, we consider MAXIMUM setting to prevent the XPA from underestimating the power consumption. The skew rate from the XPA for MAXIMUM setting is a little higher than the Wattsup? Pro, which means the XPA are less probable to underestimate the power consumption.

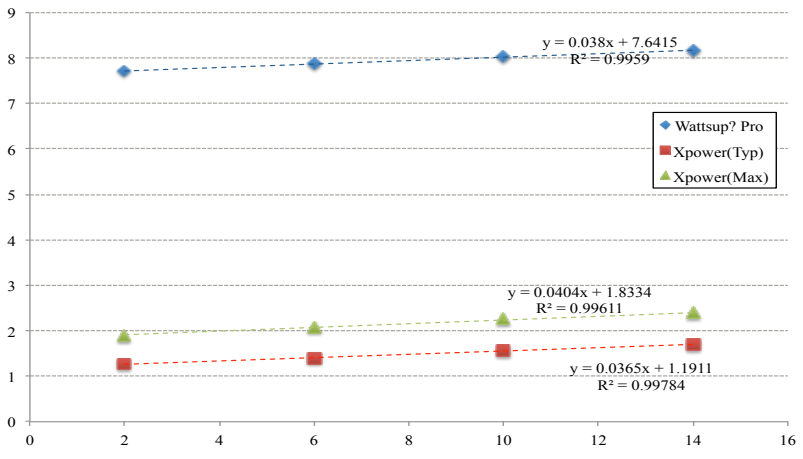

Figure 26. Power consumption estimation on the XUPV5-LX110T for step 2

Dynamic power consumptions vary according to clock frequencies. Figure 27 describes the measured power consumptions on the FPGA board using Wattsup? Pro when the operations of the step 2 are run on the Xilinx XC5VLX110T FPGA when f = {25 MHz, 50 MHz, 75 MHz, 100 MHz, 125 MHz} and the number of PEs is 14. Based on Figure 27, the power consumption is also linearly increased according to clock rates. Notice that the operational voltage is fixed for the measurement. Therefore, the power consumption increases linearly according to clock rates. If dynamic voltage scaling is employed, the power may increase qubically according to clock rates, since the required operational voltage should be increased according to clock rates as well.

Average power consumptions from the XPA and Wattsup? Pro are described in Table XII.

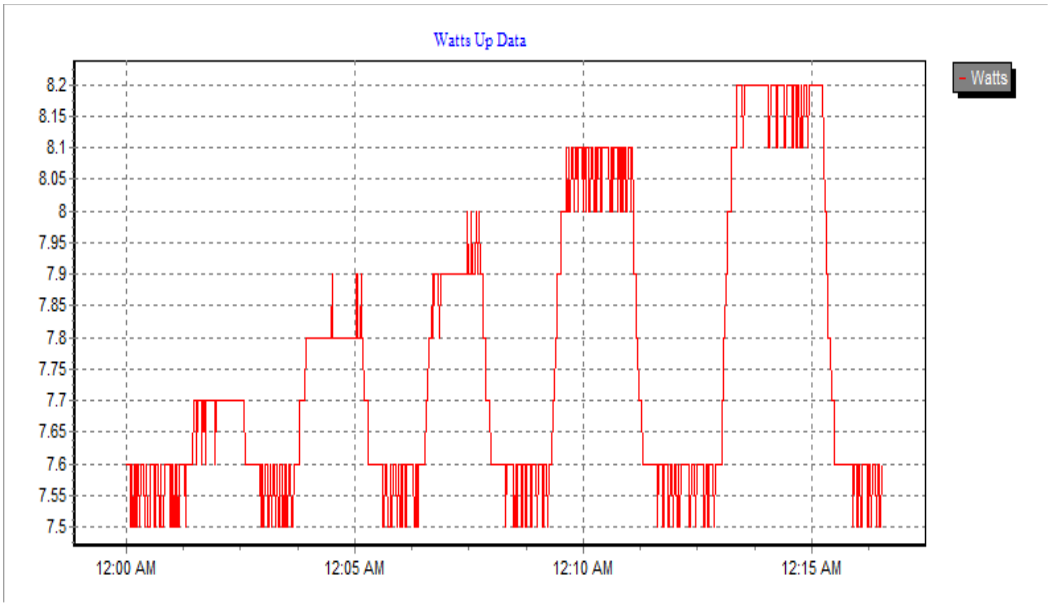

Figure 27. Power consumption measurement on the XUPV5-LX110T for step 2

TABLE XII. POWER MEASUREMENT ACCORDING TO CLOCK VARIATION

Frequency (MHz)	Wattsup? Pro (W) XUPV5-LX110T board	XPA (W) XC5VLX110T chip -Typical-	XPA (W) XC5VLX110T chip -Maximum-
25	7.69	1.239	1.905
50	7.81	1.361	2.034
75	7.90	1.481	2.161
100	8.06	1.590	2.275
125	8.17	1.695	2.387

Figure 28 represents the average dynamic power consumptions between the XPA and Wattsup? Pro. The relative error between the XPA using MAXIMUM setting and the Wattsup? Pro is within 3 %. Hence, when we estimate power consumption for step 2 on the Xilinx XC6VSX475T later, we consider the MAXIMUM setting to prevent underestimating the power consumptions

.

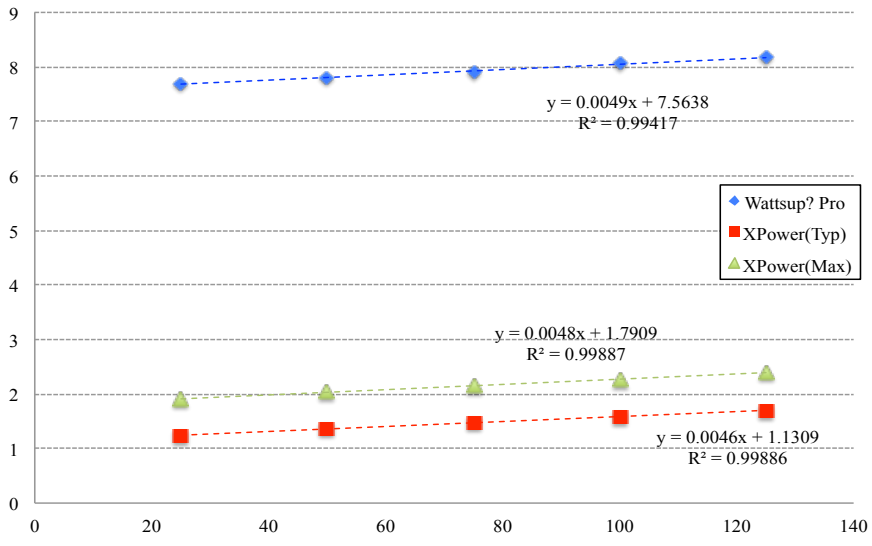

Figure 28. Power for the board and the onchip on the XUPV5-LX110T for step 2

Now, we estimate energy-based performance for the NVIDIA Tesla C2075 board and the Xilinx XUPV5-LX110T board. For the energy-based performance for the GPU, the source codes in [86] are employed to measure the power consumption on the GPU board. NVidia Management Library (NVML) is employed for the source codes. The error bound is within 5 W and applied frequency is 3.3 KHz for the measurement. The XPA 13.2 is used to estimate the power consumption on the Xilinx XUP5VLX-110T board. Average power consumptions for step 2 on the NVidia Tesla C2075 are increased up to 125 W for both single and double precision [86]. Based on Table VII, the energy-based performances for single precision step 2 on the two platforms are described as follows :

$F_{JLE\_XUP\_S}$ = 3.5 GFlops/s / 8.17 W = 428 MFlops/Joule,

$F_{JLE\_C2075\_S}$ = 62 GFlops/s / 125 W = 496 MFlops/Joule.

The FPGA board shows similar energy-based performance to the GPU. However, the XUP board consumes relatively large static power compared to dynamic power, because it is a general-purpose development board.

Now, we explore the design effectiveness for the step 2 implementations on the two platforms. First, we assume that the GPU requires around 80 W for static power consumption (c.f., 57 W for idle power consumption on the Tesla C1060). Assuming that required static power is already provided, the energy-based performances for single precision step 2 for the two platforms are represented as follows :

$F_{JLE\_D\_XUP\_S}$ = 3.5 GFlops/s / (8.17 – 7.56) W = 5.74 GFlops/Joule,

$F_{JLE\_D\_C2075\_S}$ = 62 GFlops/s / (125 – 80) W = 1.38 GFlops/Joule.

The energy-based performance on the FPGA board is better than the GPU with the assumptions, since we can obtain higher clock-based performance due to flexibility of the design choices in the FPGA. Now, we can seek the ratio of design effectiveness for the two platforms based on (6.9).

$\Lambda_{XUP}/\Lambda_{C2075} = (F_{JLE\_D\_XUP\_S}/F_{JLE\_D\_C2075\_S})\ (C_{XUP}/C_{C2075})\ (V_{XUP}/V_{C2075})^2$, where $V_{XUP} \approx 1$ and $V_{GPU} \approx 0.975$ [99]. Therefore,

$\Lambda_{XUP}/\Lambda_{GPU} \approx (5.74/1.38)\ (1/0.975)^2\ (C_{XUP}/C_{C2075}) \approx 4.38\ (C_{XUP}/C_{GPU})$.

Hence, assuming that the two effective capacitances such as C_{XUP} and C_{GPU} are similar each other, the design effectiveness for the Xilinx XC5VLX110T FPGA is around 4 times better than the GPU.

Now, we estimate the performance if a larger FPGA Xilinx XC6VSX475T is considered. We employ the XPA to estimate the power consumptions to run the operations of step 2 on the Xilinx XC6VSX475T FPGA. Since the Xilinx XC6VSX475T is a gigantic FPGA, we consider a heat sink as medium profile and airflow as 250 LFM to estimate the power consumptions on the FPGA. The XPA process is set to MAXIMUM. Table XIII shows the power consumption estimation and energy-based performance on the Xilinx XC6VSX475T FPGAs. In the table, last column represents the energy based performances calculated by the total board power, onchip power, and dynamic power each. We compare the energy-based performances between the Xilinx XC6VSX475T and NVidia Tesla C2075 GPU. We assume that the required power for the board for the FPGA is the same as XUPV5-LX100T (i.e. 7.56 W).

TABLE XIII. POWER COMSUMPTION ESTIMATION ON XILINX XC6VSX475T

Precision		Power Consumption (Stat'/Dyn')	# of PEs	PAR CLK [MHz]	Time-based Performance [GFlops/s]	Energy-based Performance Board/Onchip/Dyn' [GFlops/joule]
Exp Size	Mant' Size					
8	23(S)	13.471 (8.993 / 4.478)	170	80	27.2	1.293/2.019/6.074
11	38	13.266 (8.965 / 4.301)	106	79	16.7	0.802/1.259/3.883
11	52(D)	13.636 (9.016 / 4.620)	83	82	13.6	0.642/0.997/2.944
15	63	13.618 (9.013 / 4.605)	65	78	10.1	0.477/0.742/2.193

Considering the board powers, the energy-based performances for the implementations of single and double precision step 2 implementations on the XC6VSX475T and the Nvidia Tesla C2075 can be represented as follows :

$F_{JLE\_C2075\_S} \approx$ 496 MFlops/Joule, $F_{JLE\_XC6\_S} \approx$ 1293 MFlops/Joule,

$F_{JLE\_C2075\_D} \approx$ 248 MFlops/Joule, $F_{JLE\_XC6\_D} \approx$ 642 MFlops/Joule.

Now, we finally can compare the time-, clock-, energy-based performance between the Xilinx XC6VSX475T and NVidia Tesla C2075 for single and double precision step 2 as follows :

$P_{XC6\_S}/P_{C2075\_S} \approx 27.2/62.0 \approx 0.44$, $P_{XC6\_D}/P_{C2075\_D} \approx 13.6/31.0 \approx 0.44$,

$F_{CLK\_XC6\_S}/F_{CLK\_C2075\_S} \approx 340.0/53.9 \approx 6.31$,

$F_{CLK\_XC6\_D}/F_{CLK\_C2075\_D} \approx 165.9/27.0 \approx 6.14$,

$F_{JLE\_XC6\_S}/F_{JLE\_C2075\_S} \approx 1293/496 \approx 2.61$,

$F_{JLE\_XC6\_D}/F_{JLE\_C2075\_D} \approx 642/248 \approx 2.59$.

Hence, the time-, clock-, and energy-based performances on the FPGA are better around {0.4, 6.2, 2.6} times for single and double precision step 2 than the GPU. We can estimate design effectiveness for both platforms assuming that the GPU requires 80 W for the static power consumption. Then, the comparison for the energy-based performances based on dynamic power consumption between the GPU and the FPGA can be represented as follows :

$$F_{JLE\_D\_XC6\_S} / F_{JLE\_D\_C2075\_S} \approx 6.07/1.38 \approx 4.40,$$

$$F_{JLE\_D\_XC6\_D} / F_{JLE\_D\_C2075\_D} \approx 2.94/0.69 \approx 4.26.$$

Based on (6.9),

$$\Lambda_{XC6\_S}/\Lambda_{C2075\_S} = (4.40)\ (1/0.975)^2\ (C_1/C_2) = 4.63\ (C_1/C_2),$$

$$\Lambda_{XC6\_D}/\Lambda_{C2075\_D} = (4.26)\ (1/0.975)^2\ (C_1/C_2) = 4.48\ (C_1/C_2).$$

Finally, let us examine the impact of AIR on the time-, clock-, and energy-based performances. The time-based performances for SDIR, XMIR and AIR based on (6.3) are as follows :

$$P_{SDIR\_C2075} \approx 41\ GFLOPs,$$

$$P_{XMIR\_XC6} \approx \{27.2, 20.7, 18.1, 14.7\}\ GFLOPs,$$

$$P_{AIR\_XC6} \approx \{27.2, 20.7, 19.3, 17.5\}\ GFLOPs,$$

where {w, x, y, z} represents the time-based performances when the required mantissa bit widths for prescribed accuracies = {23, 38, 52, 65}.

Table XIV shows the three types of performances for refinement procedure per iteration in SDIR on the Nvidia Tesla C 2075, XMIR on the Xilinx XC6VSX475T FPGA, and AIR on the Xilinx XC6VSX475T FPGA according to prescribed accuracies.

TABLE XIV. PERFORMANCES FOR SDIR, XMIR AND AIR

UNITS : P_T : GFLOPS/S, F_{CLK} : FLOPS/CC, F_{JOULE} : GFLOPS/JOULE

Required Accuracy	NVidia C 2075 SDIR			XC6VSX475T XMIR			XC6VSX475T AIR		
	P_T	F_{CLK}	F_{JLE}	P_T	F_{CLK}	F_{JLE}	P_T	F_{CLK}	F_{JLE}
2^{-24}(S)	62	53.9	0.50	27.2	340.0	1.29	27.2	340.0	1.29
2^{-39}	41	35.7	0.33	20.7	258.8	0.98	20.7	258.8	0.98
2^{-53}(D)	41	35.7	0.33	18.1	226.3	0.86	19.3	241.3	0.92
2^{-64}	?	?	?	14.7	183.8	0.70	17.5	218.8	0.83

The time-, clock-, and energy-based performances on Table XIV are described in Figure 29, 30 and 31. In the figures, the x axis represents $-\log_2$ based accuracies and the y axis represents the performances according to the prescribed accuracies.

The GPU may show the significant performance drop if a user requires a higher accuracy than double precision since the GPU does not support hardware units for beyond double precision arithmetic operations. AIR on the FPGA produces around 47 % - 66 % of time-based performance compared to SDIR on the GPU when a user requires a prescribed accuracy between single and double precision. As for the clock- and energy-based performances, AIR shows around 6.8 - 9.5 times better for the clock-based performance and around 2.8 - 3.9 times better for the energy-based performance than the GPU when a user requires a prescribed accuracy between single and double precision.

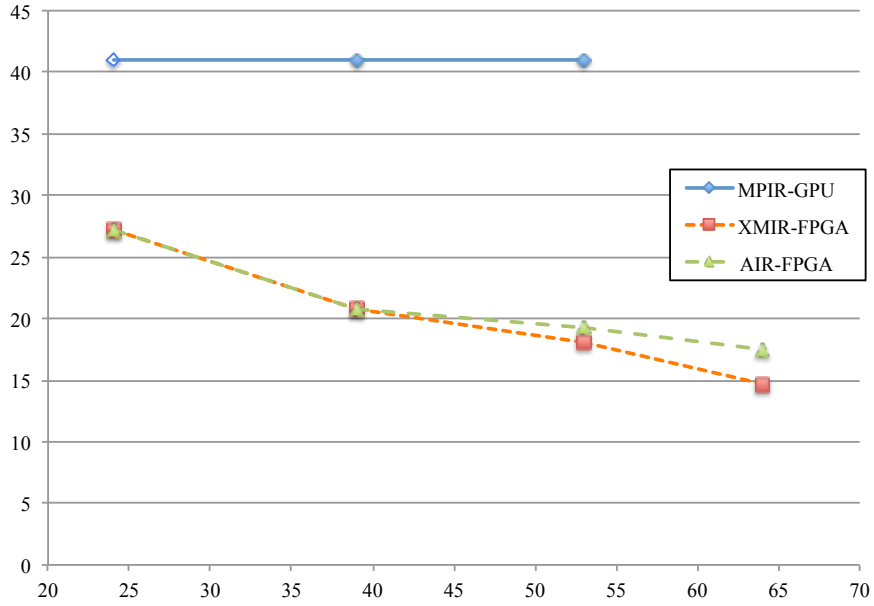

Figure 29. Time-based performance comparison

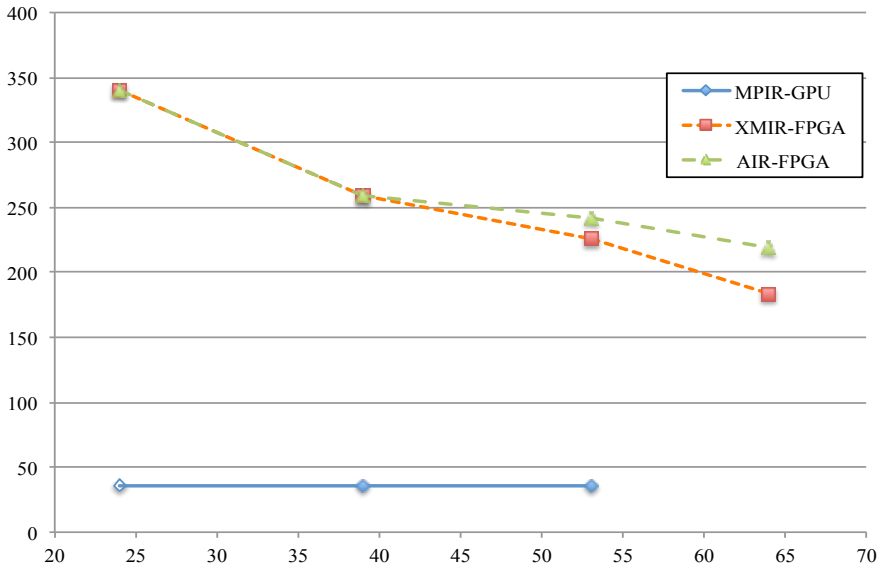

Figure 30. Clock-based performance comparison

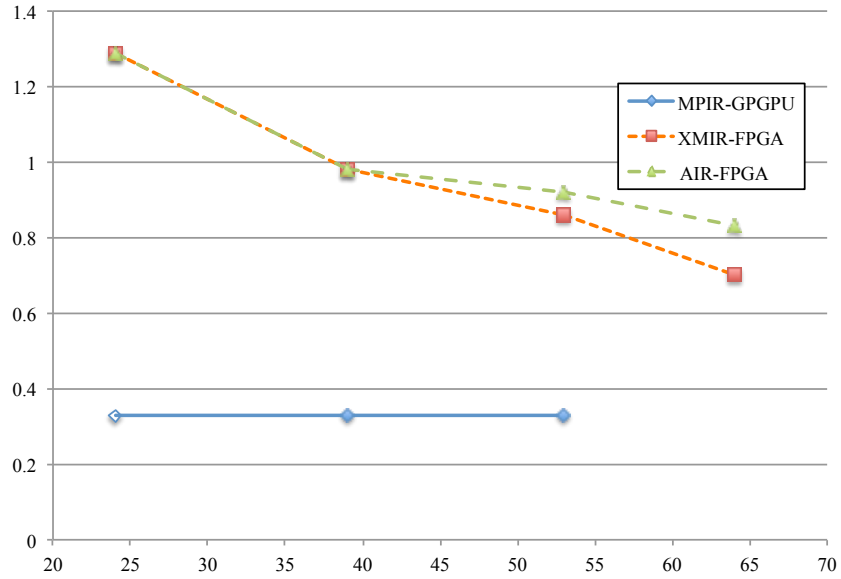

Figure 31. Energy-based performance comparison

6-6. Summary of chapter

AIR on the XC6VSX475T FPGA shows around 6.8 - 9.5 times better for the clock-based performance and around 2.8 - 3.9 times better for the energy-based performance than the SDIR on the Nvidia Tesla C2075 when a user requires a prescribed accuracy between single and double precision, while 47% - 66% in the time-based performance. When a user requires the double precision accuracy for a linear system solver, AIR shows around 7% improvement over XMIR for the time-, clock-, and energy-based performances for refinement procedure.

Generally, FPGA is energy effective while GPU is time effective. We explain energy efficiency for the FPGA by introducing the design effectiveness, which is defined by the number of transistors ratio between the number of transistors participating in

actual computation to produce floating point operations and the number of transistors participating in all switching activities. The design effectiveness for the implementation on the FPGA can be superior to the implementation on the GPU because of flexible design on the FPGA.

Chapter 7

Future work and conclusions

Utilizing precision in FPGAs can produce higher performance and lower power consumption given a prescribed accuracy. In this dissertation, the AIR algorithm is developed to produce optimized performance for linear system applications by utilizing precision *adaptively* on FPGAs. This dissertation shows that AIR on the Xilinx XC6VSX475T FPGA can produce higher design effectiveness and around 2.8 - 3.9 times better energy efficiency compared to the Nvidia Tesla C2075 GPU when a user requires accuracy between single and double precision. However, the implementation on the FPGA shows 47% - 66% of the time-based performance on the GPU.

Adaptive dynamic precision computations can be utilized in many other applications such as iterative methods and eigenvalue problems. Since this research represents the first effort to implement an adaptive dynamic precision iterative refinement for a real computing platform, this may help the study to explore adaptive dynamic precision algorithms for applications in near future. Especially, the impact of AIR on the performance becomes more significant, when prescribed solution accuracy becomes higher.

Exploring the AIR algorithm for parallel computation and tuning FPGA circuit for AIR are remained in future work.

List of References

[1] (2011). *Quote investigator*. Available: http://quoteinvestigator.com/2011/05/13/einstein-simple/

[2] R. Sessions, "How a 'Difficult' Composer Gets That Way," *New York Times,* January 8, 1950.

[3] D. H. Bailey, "High-Precision Floating-Point Arithmetic in Scientific Computation," *Computing in Science and Engg.,* vol. 7, pp. 54-61, 2005.

[4] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*: {Morgan Kaufmann}, 1998.

[5] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*: {Morgan Kaufmann}, 2002.

[6] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.,* vol. 34, pp. 171-210, 2002.

[7] J. Lee, G. D. Peterson, R. J. Hinde, and R. J. Harrison, "Mixed Precision Dense Linear System Solvers High Performance Reconfigurable Computing," presented at the Symposium on Application Accelerators in High Performance Computing, 2009.

[8] J. Lee, J. Sun, G. Peterson, R. Harrison, and R. Hinde, "Accelerator performance comparison for mixed precision linear solvers," in *IEEE Symposium on Field-Programmable Custom Cumputing Machines* 2010.

[9] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-Performance Mixed-Precision Linear Solver for FPGAs," *IEEE Trans. Comput.,* vol. 57, pp. 1614-1623, 2008.

[10] C. Moler, *Numerical Computing with MATLAB*: the Society for Industrial and Applied Mathematics, 2004.

[11] J. A. Goldstein and M. Levy, "Linear algebra and quantum chemistry," *Am. Math. Monthly,* vol. 98, pp. 710-718, 1991.

[12] J. Lee, G. Peterson, R. Harrison, and R. Hinde, "Hardware accelerated scalable parallel random number generators for Monte Carlo methods," 2008, pp. 177-180.

[13] A. Norris, *Computational Chemistry - An introduction to numerical methods*: John Wiley and Sons, 1981.

[14] A. Edelman, "Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence," *The International Journal of Supercomputer Applications,* vol. 7, pp. 113-128, 1993.

[15] G. Strang, *Linear algebra and its applications 4th edition*: Thomson Brooks/Cole, 2006.

[16] L. N. Trefethen, *Numerical Linear Algebra*: SIAM, 1998.

[17] J. Wilkinson, "Progress Report on the Automactic Computing Engine," 1948.

[18] A. Kielbasinski, "Iterative refinement for linear systems in variable-precision arithmetic," *BIT,* vol. 21, pp. 97-103, 1981.

[19] A. Smoktunowicz and J. Sokolnicka, "Binary cascades iterative refinement in doubled-mantissa arithmetics," *BIT,* vol. 24, pp. 123-127, 1984.

[20] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surv.,* vol. 18, pp. 365-396, 1986.

[21] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, "Design of cache memories for multi-threaded dataflow architecture," presented at the Proceedings of the 22nd annual international symposium on Computer architecture, S. Margherita Ligure, Italy, 1995.

[22] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation*: Morgan Kaufmann, 2007.

[23] D. A. Kramer and I. D. Scherson, "Dynamic Precision Iterative Algorithms," in *The IEEE 1992 Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[24] M. Zubair, S. N. Gupta, and C. E. Grosch, "A variable precision approach to speedup iterative schemes on fine grained parallel machines," *Parallel Computing,* vol. 18, pp. 1223-1231, 1992.

[25] *. Xilinx University Program : XUPV5-LX110T Development System*. Available: http://www.xilinx.com/univ/xupv5-lx110t.htm

[26] A. Al-Kurdi and D. R. Kincaid, "LU-decomposition with iterative refinement for solving sparse linear systems," *J. Comput. Appl. Math.,* vol. 185, pp. 391-403, 2006.

[27] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, "Error bounds from extra-precise iterative refinement," *ACM Trans. Math. Softw.,* vol. 32, pp. 325-351, 2006.

[28] N. J. Higham, "Iterative refinement enhances the stability of QR factorization methods for solving linear equations," *BIT,* vol. 31, pp. 447-468, 1991.

[29] N. J. Higham, "Iterative refinement for linear systems and LAPACK," *The Insititute of Mathematics and its Applications Journal of Numerical Analysis,* vol. 17, pp. 495-509, 1997.

[30] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*: Society for Industrial and Applied Mathematics, 2002.

[31] M. Jankowski and H. Wozniakowski, "Iterative refinement implies numerical stability," *BIT,* vol. 17, pp. 303 - 311, 1977.

[32] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," presented at the the ACM/IEEE conference on Supercomputing, Tampa, Florida, 2006.

[33] J. Lee and G. D. Peterson, "Iterative Refinement on FPGAs," in *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2011, pp. 8-13.

[34] R. D. Skeel, "Iterative refinement implies numerical stability for Gaussian elimination," *Mathematics of computation,* vol. 35, pp. 817-832, 1980.

[35] G. Stewart, *Introduction to matrix computations*: Academic press, 1973.

[36] J. Wilkinson, *Rounding errors in algebraic processes*: Prentice Hall, 1963.

[37] C. B. Moler, "Iterative Refinement in Floating Point," *J. ACM,* vol. 14, pp. 316-321, 1967.

[38] G. Stewart, *Matrix Algorithms*: Society for Industrial and Applied Mathematics, 1998.

[39] K. Prikopa, "Analysis and evaluation of Binary Cascade Iterative Refinement and comparison to other iterative refinement algorithms for solving linear systems," *Masterarbeit, Universität Wien. Fakultät für Informatik* 2011.

[40] J. Demmel, Y. Hida, W. Kahan, X. Li, and S. Mukherjee, *Error Bounds from Extra-precise Iterative Refinement*: EECS Department, University of California, Berkeley\, 2005.

[41] A. Buttari, J. Dongarra, J. Langou, P. Luszczek, and J. Kurzak, "Mixed precision iterative refinement techniques for the solution of dense linear systems," *International Journal of High Performance Computing Applications,* vol. 21, pp. 457-466, Win 2007.

[42] A. B. Marc Baboulin, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, Stanimire Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications,* 2008.

[43] J. H. Wilkinson, "Error analysis of floating-point computation," *Numerische Mathematik,* vol. 2, pp. 319-340, 1960.

[44] . *MAGMA v0.2 user guide*. Available: http://icl.cs.utk.edu/magma/

[45] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana, "Solving Dense Linear Systems on Graphics Processors," presented at the Proceedings of the 14th international Euro-Par conference on Parallel Processing, Las Palmas de Gran Canaria, Spain, 2008.

[46] D. Goddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations," *Int. J. Parallel Emerg. Distrib. Syst.,* vol. 22, pp. 221-256, 2007.

[47] J. Kurzak and J. Dongarra, "Implementation of mixed precision in solving systems of linear equations on the Cell processor: Research Articles," *Concurr. Comput. : Pract. Exper.,* vol. 19, pp. 1371-1385, 2007.

[48] A. R. Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan, "More flops or more precision? Accuracy parameterizable linear equation solvers for model predictive control," in *IEEE Symposium on Field Programmable Custom Computing Machines*, Napa, California, 2009.

[49] J. Lee, J. Sun, G. Peterson, R. Harrison, and R. Hinde, "Power-Aware Performance of Mixed Precision Linear Solvers for FPGAs and GPGPUs," presented at the Symposium on Application Accelerators in High Performance Computing, 2010.

[50] A. Akkas and M. J. Schulte, "Dual-mode floating-point multiplier architectures with parallel operations," *J. Syst. Archit.,* vol. 52, pp. 549-562, 2006.

[51] R. Parthasarathi, E. Raman, K. Sankaranarayanan, and L. N. Chakrapani, "A Reconfigurable Co-Processor for Variable Long Precision Arithmetic Using

Indian Algorithms," presented at the Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001.

[52] M. J. Schulte and J. Earl E. Swartzlander, "A Family of Variable-Precision Interval Arithmetic Processors," *IEEE Trans. Comput.,* vol. 49, pp. 387-397, 2000.

[53] A. Bojanczyk, "Complexity of Solving Linear Systems in Different Models of Computation," *SIAM Journal on Numerical Analysis,* vol. 21, pp. 591-603, 1984.

[54] G. Poole and L. Neal, "Gaussian elimination: when is scaling beneficial?," *Linear Algebra and its Applications,* vol. 162, pp. 309-324, 1992.

[55] L. Neal and G. Poole, "A geometric analysis of Gaussian elimination. II," *Linear Algebra and its Applications,* vol. 173, pp. 239-264, 1992.

[56] G. Poole and L. Neal, "A geometric analysis of Gaussian elimination. I," *Linear Algebra and its Applications,* vol. 149, pp. 249-272, 1991.

[57] P. B. Hansen, "Householder reduction of linear equations," *ACM Comput. Surv.,* vol. 24, pp. 185-194, 1992.

[58] A. S. Householder, "Unitary Triangularization of a Nonsymmetric Matrix," *J. ACM,* vol. 5, pp. 339-342, 1958.

[59] G. Poole and L. Neal, "The Rook's pivoting strategy," *J. Comput. Appl. Math.,* vol. 123, pp. 353-369, 2000.

[60] L. V. Foster, "Gaussian Elimination with Partial Pivoting Can Fail in Practice," *SIAM J. Matrix Anal. Appl.,* vol. 15, pp. 1354-1362, 1994.

[61] C. G. Broyden, "Some Condition-Number bounds for the Gaussian Elimination Process," *IMA Journal of Applied Mathematics,* vol. 12, pp. 273-286, December 1, 1973 1973.

[62] R. D. Skeel, "Scaling for Numerical Stability in Gaussian Elimination," *J. ACM,* vol. 26, pp. 494-526, 1979.

[63] L. N. Trefethen, "Three mysteries of Gaussian elimination," *SIGNUM Newsl.,* vol. 20, pp. 2-5, 1985.

[64] L. N. Trefethen and R. S. Schreiber, "Average-case stability of Gaussian elimination," *SIAM J. Matrix Anal. Appl.,* vol. 11, pp. 335-360, 1990.

[65] A. M. Turing, "Rounding-off errors in matrix processes," *The Quarterly Journal of Mechanics and Applied Mathematics* vol. 1, pp. 287-308, 1948.

[66] S. J. Wright, "A collection of problems for which Gaussian elimination with partial pivoting is unstable," *SIAM J. Sci. Comput.,* vol. 14, pp. 231-238, 1993.

[67] N. J. Higham and D. J. Higham, "Large Growth Factors in Gaussian Elimination with Pivoting," vol. 10, pp. 155-164, 1989.

[68] W. Oettli and W. Prager, "Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides," *Numerische Mathematik,* vol. 6, pp. 405-409, 1964.

[69] J. Demmel, *Applied Numerical Linear Algebra*: Society for Industrial and Applied Mathematics (SIAM), 1997.

[70] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.,* vol. 23, pp. 5-48, 1991.

[71] J. H. Wilkinson, "Some Comments from a Numerical Analyst," *J. ACM,* vol. 18, pp. 137-147, 1971.
[72] Z. Chen and J. J. Dongarra, "Condition Numbers of Gaussian Random Matrices," *SIAM J. Matrix Anal. Appl.,* vol. 27, pp. 603-620, 2005.
[73] A. Edelman, "Eigenvalues and condition numbers of random matrices," *SIAM J. Matrix Anal. Appl.,* vol. 9, pp. 543-560, 1988.
[74] Z. Z. Chen and J. Dongarra, "Numerically stable real number codes based on random matrices," *Computational Science - Iccs 2005, Pt 1, Proceedings,* vol. 3514, pp. 115-122, 2005.
[75] A. M. Tulino and S. Verd, "Random matrix theory and wireless communications," *Commun. Inf. Theory,* vol. 1, pp. 1-182, 2004.
[76] J. Hormigo, J. Villalba, and M. J. Schulte, "A Hardware Algorithm for Variable-Precision Logarithm," presented at the Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2000.
[77] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," presented at the Proceedings of the April 18-20, 1967, spring joint computer conference, Atlantic City, New Jersey, 1967.
[78] J. Lee, G. D. Peterson, R. J. Harrison, and R. J. Hinde, "Implementation of hardware-accelerated scalable parallel random number generators," *VLSI Des.,* vol. 2010, pp. 10-10, 2010.
[79] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: a scalable library for pseudorandom number generation," *ACM Trans. Math. Softw.,* vol. 26, pp. 436-461, 2000.
[80] D. B. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," presented at the Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays, Monterey, California, USA, 2009.
[81] . *GSL - GNU Scientific Library*. Available: http://www.gnu.org/software/gsl
[82] (2012). *MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 13.4, UG081(v13.4)*. Available:http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf
[83] D. Yang, G. D. Peterson, H. Li, and J. Sun, "An FPGA Implementation for Solving Least Square Problem," presented at the Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009.
[84] "Virtex-5 Family Overview," DS100 (v5.0), 2009.
[85] "Virtex-6 Family Overview," DS150 (v2.4), 2012.
[86] K. Kasichayanula, "Power Aware Computing on GPUs," *Master Thesis, University of Tennessee - Knoxville,* 2012.
[87] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM,* vol. 31, pp. 532-533, 1988.

[88] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM,* vol. 52, pp. 65-76, 2009.

[89] J. Williams, A. George, J. Richardson, K. Gosrani, and S. Suresh, "Computational Density of Fixed and Reconfigurable Multi-Core Devices for Application Acceleration," in *Reconfigurable Systems Summer Institute (RSSI)*, Urbana, IL, 2008.

[90] J. Williams, A. George, J. Richardson, K. Gosrani, and S. Suresh, "Fixed and Reconfigurable Multi-Core Device Characterization for HPEC," in *High-Performance Embedded Computing Workshop (HPEC)*, Lexington, MA, 2008.

[91] P. Abusaidi, M. Klein, and B. Philofsky, "Virtex-5 FPGA System Power Design Consideration," WP285, 2008.

[92] S. Hong and H. Kim, "An integrated GPU power and performance model," presented at the Proceedings of the 37th annual international symposium on Computer architecture, Saint-Malo, France, 2010.

[93] NVIDIA Tesla C2075 [Online].

[94] L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic power consumption in Virtex-II FPGA family," presented at the Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, Monterey, California, USA, 2002.

[95] . *Watts up?* Available: https://http://www.wattsupmeters.com/secure/index.php

[96] "Xilinx ML505/ML506/ML507 Evaluation Platform User Guide " UG347, 2011.

[97] (2010). *Xilinx Power Tools Tutorial UG733 (v1.0)*. Available: http://www.xilinx.com/products/design_resources/power_central/

[98] (2010). *Xilinx Power Estimator 12.1*. Available: http://www.xilinx.com/products/design_resources/power_central/

[99] (2011). *Tesla C2075 computing processor board*. Available: http://www.nvidia.com/docs/IO/43395/BD-05880-001_v02.pdf

Vita

JunKyu Lee was born in Seoul, South Korea on March 9, 1974. He attended to HanYang University and received double B.S. in Physics and in Electrical Engineering in 2002. From 1995 to 1998, he served for the Korean Air Force. He worked for the LG electronics Inc. in 2002 and for the NARA Controls Inc. from 2003 to 2004. He joined the Electrical and Computer Engineering department in the University of Tennessee, Knoxville in August 2005 and received M.S. in Electrical Engineering in 2007.